UNIVERSIDAD POLITÉCNICA DE MADRID

SINTETIZADOR ANALÓGICO EN ARDUINO

JAIME CABALLERO INSAURRIAGA

SEPTIEMBRE DE 2013

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN

Información de proyecto fin de carrera

Tema: Sintetizador analógico en Arduino

Título: Sintetizador analógico en Arduino

Autor: Jaime Caballero Insaurriaga

Titulación: Ingeniería técnica de telecomunicación, especialidad en sonido e imagen

Tutor: Lino García Morales

Departamento: DIAC

Director: Lino García Morales

Miembros del tribunal:

Presidente: Gregorio Rubio Cifuentes

Vocal: Lino García Morales

Vocal Secretario: Francisco Javier Tabernero Gil

Fecha de Lectura: Septiembre de 2013

SINTETIZADOR ANALÓGICO EN ARDUINO

Este proyecto consiste en el diseño y construcción de un sintetizador basado en el chip *6581 Sound Interface Device (SID)*. Este chip era el encargado de la generación de sonido en el *Commodore 64*, ordenador personal comercializado en 1982, y fue el primer sintetizador complejo construido para ordenador. El chip en cuestión es un sintetizador de tres voces, cada una de ellas capaz de generar cuatro diferentes formas de onda. Cada voz tiene control independiente de varios parámetros, permitiendo una relativamente amplia variedad de sonidos y efectos, muy útil para su uso en videojuegos. Además está dotado de un filtro programable para conseguir distintos timbres mediante síntesis sustractiva. El sintetizador se ha construido sobre Arduino, una plataforma de electrónica abierta concebida para la creación de prototipos, consistente en una placa de circuito impreso con un microcontrolador, programable desde un PC para que realice múltiples funciones (desde encender LEDs hasta controlar servomecanismos en robótica, procesado y transmisión de datos, etc.).

El sintetizador es controlable vía MIDI, por ejemplo, desde un teclado de piano. A través de MIDI recibe información tal como qué notas debe tocar, o los valores de los parámetros del SID que modifican las propiedades del sonido. Además, toda esa información también la puede recibir de un PC mediante una conexión USB. Se han construido dos versiones del sintetizador: una versión “hardware”, que utiliza el SID para la generación de sonido, y otra “software”, que reemplaza el SID por un emulador, es decir, un programa que se comporta (en la medida de lo posible) de la misma manera que el SID. El emulador se ha implementado en un microcontrolador *Atmega 168* de *Atmel*, el mismo que utiliza Arduino.

ANALOG SYNTHESIZER ON ARDUINO

This project consists on design and construction of a synthesizer which is based on chip *6581 Sound Interface Device (SID)*. This chip was used for sound generation on the *Commodore 64*, a home computer presented in 1982, and it was the first complex synthesizer built for computers. The chip is a three-voice synthesizer, each voice capable of generating four different waveforms. Each voice has independent control of several parameters, allowing a relatively wide variety of sounds and effects, very useful for its use on videogames. It also includes a programmable filter, allowing more timbre control via subtractive synthesis. The synthesizer has been built on Arduino, an open-source electronics prototyping platform that consists on a printed circuit board with a microcontroller, which is programmable with a computer to do several functions (lighting LEDs, controlling servomechanisms on robotics, data processing or transmission, etc.).

The synthesizer is controlled via MIDI, in example, from a piano-type keyboard. It receives from MIDI information such as the notes that should be played or SID's parameter values that modify the sound. It also can receive that information from a PC via USB connection. Two versions of the synthesizer have been built: a hardware one that uses the SID chip for sound generation, and a software one that replaces SID by an emulator, it is, a program that behaves (as far as possible) in the same way the SID would. The emulator is implemented on an *Atmel's Atmega 168* microcontroller, the same one that is used on Arduino.

TABLA DE CONTENIDO

1. INTRODUCCIÓN

En el año 1982 la compañía *Commodore International* lanzó al mercado el ordenador de 8 bits *Commodore 64* (también conocido como *C64*). Este ordenador tuvo gran éxito comercial en su momento, en buena parte debido sus posibilidades gráficas y de sonido, superiores a las de sus predecesores y coetáneos, así como a su reducido precio. Era un momento en que el éxito en el mercado de los llamados *home computers* ("ordenadores domésticos") dependía mucho de la industria de los videojuegos, y el C64 se diseñó apuntando mucho hacia esa industria.

En lo que a este proyecto interesa, el sonido, el C64 incluía un chip destinado exclusivamente a ello: el *6581 Sound Interface Device* (más conocido por sus siglas *SID, o 6581*). El chip en cuestión es un sintetizador de tres voces, cada una de ellas capaz de generar cuatro diferentes formas de onda. Además cada voz tiene control independiente de varios parámetros, permitiendo una relativamente amplia variedad de sonidos y efectos, muy útil para su uso en videojuegos (MacGateway, 2012).

El SID tiene cierta importancia histórica, pues se trata del primer sintetizador diseñado para ordenador capaz de generar sonidos complejos. Aunque otros modelos de ordenadores ya eran capaces de generar sonido, como los *IBM PC* o el *Apple II*, sus posibilidades eran muy limitadas, y su calidad muy baja. El IBM PC incluía el llamado PC Speaker, un altavoz piezoeléctrico capaz de generar un único zumbido a una determinada frecuencia (MacGateway, 2012). El *Apple II* incluía un circuito tipo interruptor conectado a un altavoz, que permitía generar simples chasquidos o *clicks*. Por ejemplo, para generar un sonido de 440 Hz era necesario producir por software 440 *clicks* por segundo (Sawsquarenoise, 2011). Poco después de la comercialización del SID los distintos modelos de ordenadores comenzaron a incluir chips y tarjetas de expansión que igualaban, e incluso mejoraban las posibilidades del SID, que fue pionero en la materia.

A día de hoy (y desde hace muchos años) el 6581 no se fabrica, sin embargo sigue estando valorado entre músicos nostálgicos y amantes de los ordenadores antiguos. Aunque la calidad del sonido es razonablemente buena, no es de los sintetizadores de más calidad ni mayores posibilidades, pero su sonido característico tiene muchos seguidores. De hecho, todavía se sigue componiendo música para el SID, y se tienen muy en cuenta a los compositores "clásicos" que compusieron las melodías de conocidos videojuegos del C64.

1.1 Objetivos

El objetivo principal de este proyecto es el diseño y construcción del prototipo de un sintetizador basado en el chip *6581 Sound Interface Device* (más conocido por sus siglas *SID, o 6581*), encargado de la generación de sonido del ordenador Commodore 64. Dicho sintetizador puede funcionar en modo monofónico (una sola nota simultánea) o polifónico (hasta un máximo de tres notas simultáneas, debido a

la limitación del SID). Además, permite modificar los parámetros del sonido, y guardar la configuración obtenida en forma de presets.

Se pretende que el sintetizador se pueda utilizar como un instrumento musical, por ejemplo, desde un teclado de piano, para lo cual se utiliza el estándar MIDI. A través de MIDI recibe información tal como qué notas debe tocar, o los valores de los parámetros del SID que modifican las propiedades del sonido. Además, toda esa información también la puede recibir de un PC mediante una conexión USB.

Se han construido dos prototipos: una versión "hardware", que utiliza el SID para la generación de sonido, y otra "software", que reemplaza el SID por un emulador, es decir, un programa que se comporta (en la medida de lo posible) de la misma manera que el SID. El diseño de una versión software tiene sentido, pues el 6581 ya no se fabrica, y es relativamente difícil encontrarlo en el mercado de segunda mano, y más en buenas condiciones, además de resultar caro. Sin embargo, la versión software se ha implementado en un microcontrolador *Atmega 168*, que resulta muy económico (un microcontrolador de este modelo puede costar entre 5 y 4 €).

El sistema escogido para construir el sintetizador es Arduino, una plataforma de electrónica abierta para la creación de prototipos. Arduino es una plataforma basada en una placa de circuito impreso con un microcontrolador, que es fácilmente programable desde un PC para que realice múltiples funciones (desde encender LEDs hasta controlar servomecanismos en robótica, o procesado y transmisión de datos, etc.). El principal motivo del uso de Arduino en este proyecto es el bajo coste que tiene esta plataforma: una placa Arduino básica puede costar del orden de 20 €, y el software necesario para programarla se puede descargar gratuitamente del sitio web oficial (Arduino).

Otro de los aspectos interesantes relacionados con Arduino es la filosofía "Do it yourself" (DIY, hágalo usted mismo). Así, cualquier persona con una placa Arduino y un PC puede construir sus sistemas electrónicos, simplemente descargando el código del programa y construyendo una placa de circuito. Además, como es abierto, también se puede modificar sus funcionalidades. También para este proyecto resulta interesante aplicar un poco de ese enfoque, para que con instrucciones sobre la construcción de los circuitos y el código del programa sea suficiente para que cualquier persona pueda construir el sintetizador.

1.2 Síntesis sustractiva

En el contexto de los instrumentos musicales, se llama sintetizador a un dispositivo electrónico (o un software) capaz de generar sonido. De forma más precisa, un sintetizador genera una señal eléctrica destinada a ser convertida en sonido a través de un altavoz. Existen diversas técnicas de síntesis (aditiva, FM, etc.), pero aquí se explican las bases de la síntesis sustractiva, pues es la que utiliza el SID. Para ello, primero conviene explicar en qué consiste el timbre de los sonidos, para luego especificar forma en que la síntesis sustractiva consigue sonidos de distinto timbre.

Se puede definir el timbre como la cualidad de un sonido que permite distinguir distintos instrumentos musicales interpretando la misma nota a la misma intensidad, o lo que es lo mismo, distinguir sonidos de igual frecuencia fundamental. En ese sentido, el sonido más simple es el tono puro, que se corresponde con la expresión de la ecuación (1.1):

$$x(t) = \cos(2\pi f t), \tag{1.1}$$

siendo f la frecuencia del sonido, y t el tiempo en segundos. Por el desarrollo en serie de Fourier cualquier señal periódica puede entenderse como una suma de tonos puros de distintas frecuencias, según la expresión:[1]

$$x(t) = a_0 + 2\sum_{n=1}^{\infty} A_n \cdot \cos(2\pi n f_0 t + \theta_n) \tag{1.2}$$

En esta ecuación a_0 es la componente continua de la señal, n el número de armónico, A_n la intensidad con que aparece cada armónico, f_0 la frecuencia fundamental y θ_n la fase del armónico (Allan V. Oppenheim, 1998). De esta ecuación se saca en claro que cualquier sonido complejo se puede expresar como un conjunto de tonos puros llamados armónicos, todos ellos múltiplo de una frecuencia fundamental.

La frecuencia fundamental es la que nos da información sobre lo que en música se llama "tono" o "altura", es decir, la nota se está interpretando. La proporción entre la intensidad de los armónicos, es decir, el valor de A_n para cada uno, es una de las principales características que determinan el timbre, mientras que ni la fase θ ni la componente continua a_0 tienen un efecto apreciable. A modo de ejemplo, si un violín o una flauta interpretan un *la4* (justo el La por encima del *do* central), ambos sonidos tendrán la misma frecuencia fundamental, 440 Hz, pues tienen la misma altura. Sin embargo, la proporción entre la intensidad de los armónicos es diferente, y se percibirán por ello con distinto timbre. Siendo que la intensidad relativa entre los armónicos es el principal parámetro que determina el timbre, para conseguir distintos sonidos hay que tener una forma de conseguir distintas proporciones de armónicos.

La idea de la síntesis sustractiva consiste en generar sonidos que ya son de por sí complejos, y generalmente ricos en armónicos, y, mediante filtrado, eliminar, atenuar o acentuar ciertos armónicos, modificando su timbre. Se puede entender que se consigue el sonido final "esculpiendo" un sonido dado (o más bien su espectro), eliminándole componentes hasta conseguir el resultado deseado.

[1] En realidad ésta es una particularización del desarrollo en serie de Fourier, para señales reales. Como las señales que se manejarán corresponden a sonidos, es decir, variaciones de presión, o señales eléctricas, nunca tienen valores complejos y esta expresión es válida.

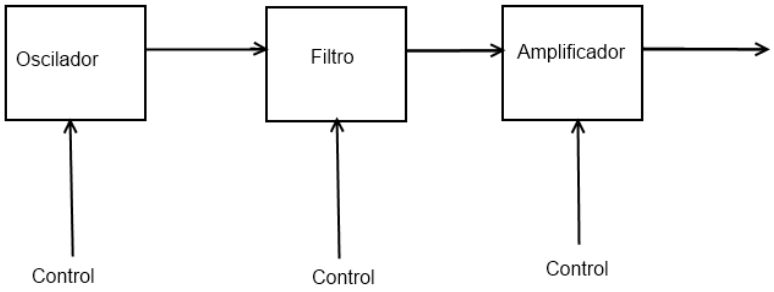

Figura 1.1. Diagrama de bloques simplificado de un sintetizador por síntesis sustractiva.

En la Figura 1.1 se muestra un modelo simplificado de un sintetizador básico de síntesis sustractiva. El oscilador es el encargado de generar la señal "en bruto", que se caracteriza por su forma de onda. Entre las formas de onda típicas se encuentran el tono puro, el diente de sierra (Figura 1.2), la onda triangular (Figura 1.3), y la onda cuadrada (Figura 1.4). El oscilador admite una señal de control, que determina la frecuencia del sonido. Como se puede observar, cada forma de onda tiene su espectro característico. Así, se tiene un control sencillo sobre el timbre del instrumento, cambiando la forma de onda generada por el oscilador.

El tono puro está compuesto únicamente por el primer armónico, el fundamental, y tiene un sonido muy "limpio", semejante al de un silbido. La onda en diente de sierra (Figura 1.2) es rica en armónicos, y tiene un sonido muy brillante. Se utiliza a menudo para emular instrumentos de viento-metal, tales como trompetas. La onda triangular (Figura 1.3) tiene pocos armónicos, y sólo contiene armónicos impares (n impar). Su timbre es meloso, suave y oscuro, y es frecuente utilizarla para conseguir sonido de tipo flauta. La onda cuadrada (Figura 1.4) es rica en armónicos, pero sólo contiene armónicos impares. Tiene un sonido brillante y hueco. Además es frecuente modular la anchura de pulso, la relación entre el tiempo que está a nivel alto y el periodo del sonido. En este caso la onda es de pulso rectangular. Al cambiar la anchura (con respecto a la onda cuadrada), comienzan a aparecer armónicos pares, y los armónicos más agudos pierden intensidad, y esto se traduce en un sonido más nasal, que puede asemejarse al de instrumentos de lengüeta como la armónica.

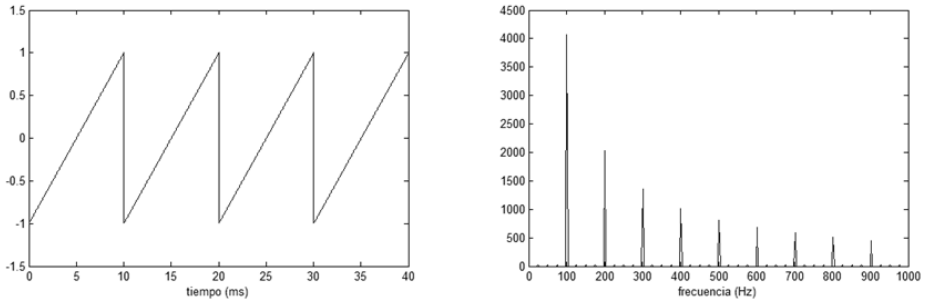

Figura 1.2. Onda en diente de sierra de 100 Hz. A la izquierda representación en tiempo, a la derecha representación en frecuencia

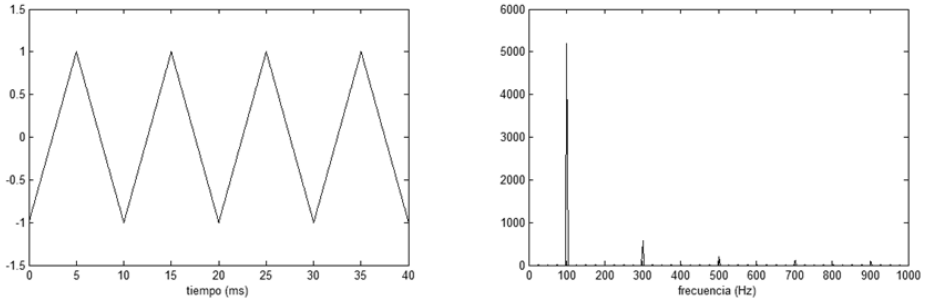

Figura 1.3. Onda triangular de 100 Hz. A la izquierda representación en tiempo, a la derecha representación en frecuencia

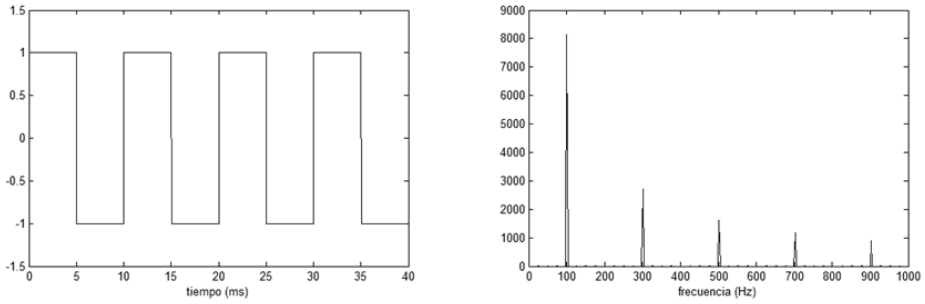

Figura 1.4. Onda cuadrada de 100 Hz. A la izquierda representación en tiempo, a la derecha representación en frecuencia

También es habitual utilizar, además, algún generador de ruido, típicamente blanco o rosa, cuya característica se puede modificar también con el filtro. El generador de ruido es útil para generar efectos de sonido tales como explosiones, disparos, o de percusión, así como se puede combinar con las otras formas de onda. Por ejemplo, para determinados sonidos de percusión se suele combinar el ruido con la forma de onda cuadrada.

La señal generada por el oscilador se pasa a través de un filtro, cuya función es atenuar, acentuar, o incluso eliminar determinados armónicos de la señal, modificando así la proporción entre ellos, y por lo tanto el timbre del sonido final. Es en este elemento, por lo tanto, donde realmente se produce la síntesis sustractiva. La frecuencia de corte es variable mediante una señal de control externa y, de hecho, lo interesante es varíe dinámicamente, estando relacionada con la frecuencia del oscilador. Si el parámetro

fuera fijo, sería más propio hablar de un control de tono que de síntesis sustractiva. Además de la frecuencia de corte, se suelen poder modificar otros parámetros del filtro, como el tipo de filtro (paso bajo, paso alto, paso banda, banda eliminada), o la resonancia, que consiste en un aumento de la ganancia a la frecuencia de corte (para paso bajo y alto).

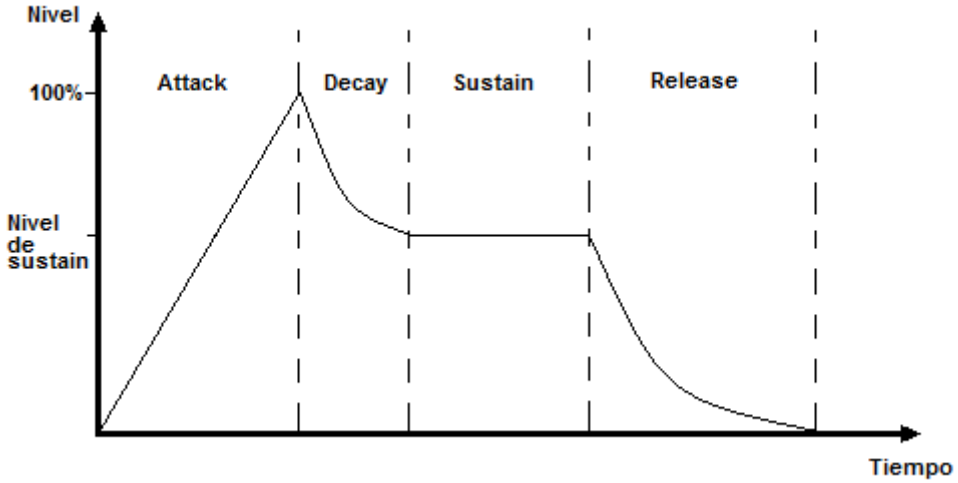

Figura 1.5. Curva ADSR

Por último se suele utilizar un amplificador de ganancia variable, para modificar dinámicamente la intensidad del sonido con la interpretación de cada nota, permitiendo más expresividad. El modelo más habitual para conseguir esto es la curva ADSR (siglas de *attack*, *decay*, *sustain*, *release*, que podrían traducirse como ataque, caída, sostenimiento y relajación), que es la forma que toma la señal de control del amplificador, y se ve en la Figura 1.5. Se trata de un modelo simplificado de la evolución natural de la intensidad en un instrumento real, y consta de cuatro etapas: al comenzar la nota (el ataque, por ejemplo al comenzar a frotar el arco de un violín) el volumen aumenta rápidamente hasta llegar al máximo, para luego caer hasta llegar a un nivel determinado, al que se mantiene hasta que el intérprete decide terminar la nota. A partir de ese momento la intensidad del sonido empieza a caer a una determinada velocidad hasta apagarse del todo. Así, se pueden obtener muchas formas de articular cada nota simplemente modificando estos cuatro parámetros: tiempo de ataque, tiempo de caída, tiempo de relajación, y nivel de sostenimiento. También se pueden obtener con este modelo algunas curvas muy diferentes a las de cualquier instrumento real.

Las posibilidades con esta simple configuración son muy variadas, simplemente controlando el origen de las señales de control. Por ejemplo, es habitual hacer modulaciones con osciladores de baja frecuencia (LFOs), añadiendo la señal que generan a las señales de control de los distintos bloques. Así, variando la frecuencia del oscilador se consigue un vibrato, variando la frecuencia de corte de un filtro paso bajo se tiene un efecto de *wah-wah*, o variando la ganancia del amplificador se obtiene un efecto de trémolo. Otra posibilidad podría ser aplicar el generador de la curva ADSR, además de al amplificador, a la frecuencia de corte del filtro, variando el timbre dinámicamente.

1.3 Estado del arte

No es este el primer intento de construir un sintetizador basado en el SID, anteriormente ha habido muchos otros proyectos que han realizado distintas aproximaciones. El propio diseñador del chip, Robert Yannes, reconoce que su intención era que el SID pudiera utilizarse en sintetizadores profesionales. Sin embargo esto nunca sucedió, pues el Commodore 64 absorbió toda la producción del chip (Varga, 1996). Todos los intentos de comercializar productos basados en el SID han tenido poca duración, pues tenían que construirse con chips de segunda mano, y cada vez quedan menos y son más caros.

Una forma de aproximación al SID es utilizarlo para interpretar música compuesta o arreglada específicamente para el chip. Esta música es capaz de explotar al máximo sus posibilidades, y aunque no es el objetivo de este proyecto (se pretende construir un instrumento, no un intérprete de música programada), cabe mencionarla. La música hecha para el SID se almacena en unos archivos llamados comúnmente "ficheros sid" "melodías sid" (*sid files* o *sid tunes*), con extensión "*.sid"*. Estos ficheros contienen el código que debería ejecutar el C64 para reproducir la composición[1]. Por lo tanto, la interpretación de música para el SID pasa por "reproducir" estos ficheros, ya sea a través del propio SID, o mediante un emulador software.

Un ejemplo de este enfoque es *HardSID*, que funciona como una tarjeta de sonido externa para ordenador, y según el modelo puede incluir de uno a cuatro SIDs. Se conecta al ordenador mediante una conexión USB, y mediante software especializado es capaz de interpretar ficheros SID (HardSID, 1999). En cuanto a la emulación software del chip, hay distintas posibilidades, pero quizás la más extendida sea la aplicación *SID player*, que cuenta con versiones para distintos sistemas operativos. Sin embargo, aunque esta aproximación es interesante por otros motivos, se aleja un poco de la filosofía de este proyecto.

SIDstation, del fabricante Elektron, es un ejemplo sintetizador en que el sonido está generado enteramente por un chip SID, y está concebido para ser utilizado como instrumento vía MIDI. Los parámetros del SID y del sintetizador son controlables mediante un teclado numérico y cuatro mandos giratorios con una pantalla LCD de 24 caracteres. Adicionalmente los parámetros también son controlables vía MIDI mediante mensajes *control change*. Este sintetizador es bastante versátil, pues además del SID en sí incluye otras posibilidades como un arpegiador para tocar automáticamente secuencias de notas, cuatro LFOs para modular distintos parámetros (frecuencia de las voces, frecuencia de corte del filtro, etc.) y la posibilidad de cambiar dinámicamente la forma de onda. Una revisión más detallada de sus características se puede encontrar en (Trask, 1999). Sin embargo, en la actualidad no se

[1] Es interesante en este sentido mencionar el proyecto *HVSC* (*High Voltage Sid Collection*), que pretende recopilar toda la música existente para el SID, desde la época del C64 hasta la actualidad. En este momento lleva recopilados más de 43000 ficheros SID, disponibles para descarga en el sitio web oficial de HVSC (HVSC).

comercializa (por la dificultad de encontrar chips) y, a pesar de sus amplias posibilidades, tampoco era una opción económica para manejar el SID.

En cuanto al diseño de un sintetizador accesible fácilmente y de precio asequible, se van a mencionar dos proyectos, ambos construidos sobre Arduino, en los cuales además se apoya el sintetizador construido. El primero es un proyecto llamado *SIDaster*, documentado en el sitio web del diseñador (Banson), que es el punto de partida para este proyecto. Se trata de un sistema diseñado para manejar el SID desde un teclado MIDI. Es un proyecto planteado con la filosofía *DIY*, y tanto el diseño del circuito como los programas Arduino necesarios se pueden descargar gratuitamente, además de estar explicado su funcionamiento. El sintetizador consta de una placa Arduino que incluye la "lógica" del sintetizador, que es quien decide cómo reaccionar a las acciones sobre el teclado, y está conectado a una placa que incluye el SID que genera el sonido. En cuanto a las características del sintetizador, es bastante simple: funciona en modo monofónico, y todos los parámetros del SID son configurables mediante mensajes *control change*. Incluye control de afinación independiente para cada voz, lo cual es útil para producir sonidos complejos. En cuanto a la conectividad, es posible recibir mensajes de un teclado o del ordenador vía USB, pero a través de la misma conexión, pudiendo provocar fallos si ambos tratan de enviar mensajes al mismo tiempo.

El objetivo del proyecto *SIDuino*, documentado en el sitio web del diseñador (Haberer), es emular por software el SID, que es difícil de conseguir, además de caro, y poder contar además con la interfaz y programabilidad de Arduino (Arduino playground). El emulador consiste en un microcontrolador Atmega 168 con un programa que imita el comportamiento del SID, conectado a la placa Arduino para controlarlo. La última versión del emulador, *SIDuino I2C*, permite control vía MIDI, funcionando como un sintetizador polifónico sencillo de tres voces. Este emulador es el punto de partida para el diseño de la versión software.

2. INTRODUCCIÓN A ARDUINO

En este apartado se da una visión básica y general de lo que es y de cómo funciona Arduino, la plataforma en que está construido el sintetizador, para poder comprenderlo con más facilidad. No se pretende dar una explicación excesivamente profunda, ya que hay abundante documentación y numerosos tutoriales disponibles en el sitio web oficial de Arduino.

Arduino, como ya se indicó, es una plataforma de electrónica abierta concebida para la creación de prototipos. Consiste en una placa con un microcontrolador, el cual se programa desde el ordenador mediante el software Arduino, conectando la placa al ordenador por USB. El lenguaje de programación utilizado es propio, y está basado en *Wiring*, que a su vez está basado en *Processing*, y la sintaxis es igual que la de *C++*. Está pensado para que sea flexible y fácil de usar (no necesariamente con conocimientos avanzados de electrónica o programación) y está teniendo una amplia difusión, debido a la gran cantidad de tutoriales y proyectos desarrollados y documentados en la web, a su bajo coste y facilidad de uso.

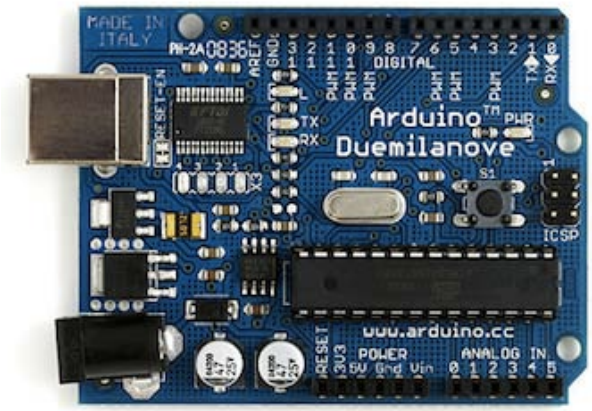

Figura 2.1. Placa Arduino Duemilanove

2.1 El microcontrolador y la placa

La placa Arduino básica, Arduino Duemilanove, está centrada en los microcontroladores *Atmega168* o *Atmega328*, del fabricante *Atmel* (ambos son esencialmente iguales, difieren en el tamaño de las memorias). Además incluye una interfaz USB, que utiliza para comunicarse con un ordenador y obtener alimentación eléctrica. La función de la placa es permitir la programación del microcontrolador, mediante conexión USB o ICSP (*In circuit serial programming*), y proporcionar un fácil acceso a todos sus terminales, mediante los pines laterales de la placa. A través de estos terminales puede o bien leer señales del exterior, que pueden proceder de algún tipo de sensor, por ejemplo de temperatura, o generar señales de control, por ejemplo para motores o leds.

Un microcontrolador es un sistema basado en un procesador, que está construido en un solo chip, y que puede usarse para desempeñar tareas muy diversas. Los modelos usados en la placa básica Arduino son en esencia iguales, y, simplificando mucho, consisten en:

- Un procesador
- Una memoria flash en la que se almacena el programa que ejecuta el procesador
- Una memoria SRAM donde se guardan las variables del programa
- Una memoria EEPROM, para almacenar datos que perduren al apagar el sistema
- Puertos de entrada/salida, conectados directamente a los terminales del chip
- Módulos de comunicaciones, para transferencia de datos según distintos protocolos: USART, SPI y TWI (*Two wire Interface*, más frecuentemente llamado I^2C)
- Módulos contadores, para funciones de temporización o PWM (modulación por anchura de pulso, un método de conversión D/A)
- Conversores analógico/digital

Los puertos de entrada/salida ocupan todos los pines del chip, salvo los de alimentación y tierra. Cuando un pin funciona como entrada/salida, se puede configurar como entrada, permitiendo al procesador leer el valor digital de la tensión en ese pin, o como salida, para que el procesador pueda controlar el nivel de tensión del pin escribiéndole un valor digital (0 para 0V, 1 para 5V).

El resto de los módulos nombrados para funcionar también pueden necesitar utilizar algunos pines del chip, así que, cuando estén en funcionamiento, esos pines no podrán usarse como entrada/salida (además hay un pin que no suele usarse, pues tiene función de reset). Por ejemplo, viendo la Figura 2.1, los pines 0 y 1 se pueden usar como entrada/salida, pero también corresponden al receptor y al transmisor de la *USART*. Así, cuando se utiliza la USART, dichos pines están configurados como receptor y transmisor de dicha interfaz, y no como entrada/salida. Se puede encontrar información más detallada en las hojas de características de los microcontroladores Atmega (Atmel Corporation, 2009).

En ocasiones (concretamente en este proyecto) un pin del microcontrolador puede ser referenciado de distintas formas: el número de pin del chip, el número de pin correspondiente de la placa Arduino, o la función que dicho pin realiza (por ejemplo, *Tx* de la *USART*). El motivo es que a veces se programa el microcontrolador fuera del contexto de Arduino, como es el caso del emulador del SID en este proyecto. Por ello, un esquema como el de la Figura 2.2 es una herramienta de gran utilidad para trabajar con estos microcontroladores.

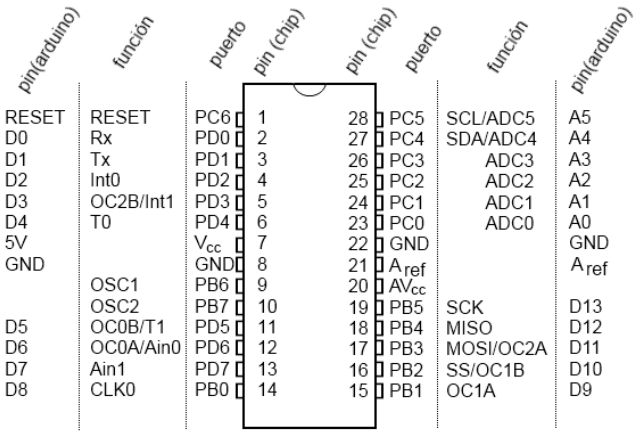

Figura 2.2. Referencia de los pines del Atmega 168 (DIP) y Arduino

2.2 El entorno de desarrollo

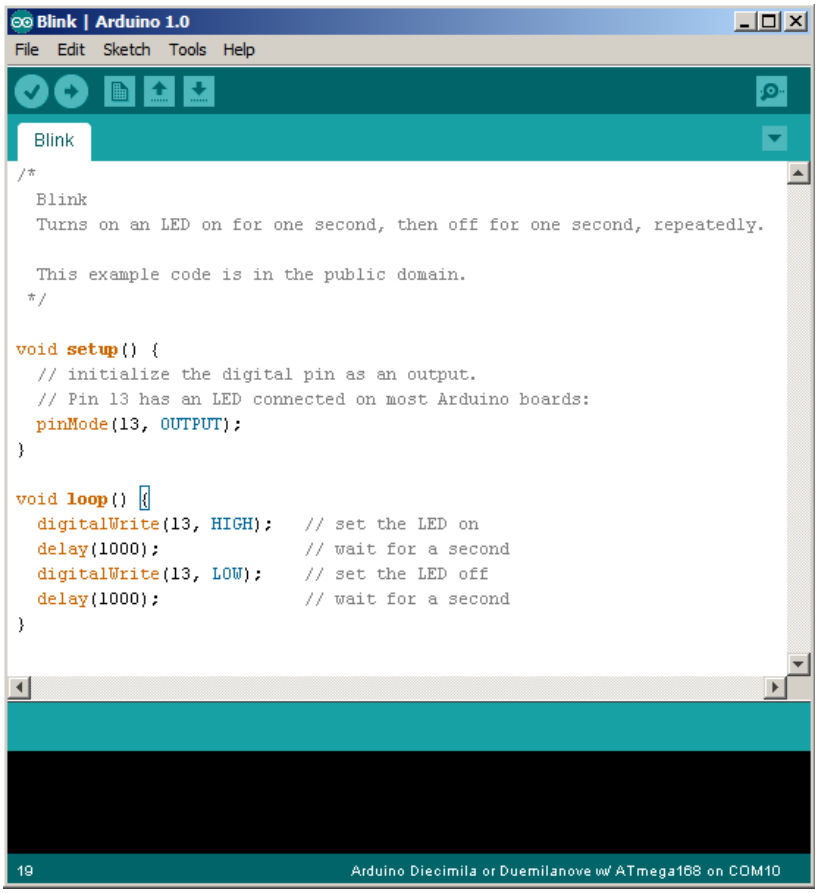

Figura 2.3. Ventana del entorno de desarrollo Arduino

En la Figura 2.3 se muestra la ventana del entorno de desarrollo (*IDE, Integrated develping environment*) de Arduino. Este es el software de ordenador que se usa para escribir los programas que se ejecutarán en la placa, y para compilar y transmitir dichos programas al microcontrolador.

En el editor de texto se escriben los programas de Arduino, que comúnmente se llaman *sketches*. En la figura se muestra un programa básico de Arduino, *blink* (este programa podría entenderse como el "Hola mundo" de Arduino), que simplemente enciende y apaga periódicamente un led conectado al pin 13 de Arduino. Se puede apreciar la simplicidad del lenguaje de programación.

Un *sketch* de Arduino siempre tiene dos funciones principales: *setup*, que se ejecuta una única vez al encender el sistema y se utiliza para configurar el micro y dar valores iniciales a las variables, y *loop*, que se ejecuta una y otra vez, en bucle, después de *setup*. En realidad, el lenguaje de programación Arduino es básicamente *C++* con gran cantidad de librerías, funciones, objetos, etc. que facilitan su uso para controlar las funcionalidades de los microcontroladores Atmega de las placas Arduino.

Una vez terminado el programa, subir el código al Atmega consiste en conectar la placa por USB al ordenador, y desde la IDE seleccionar *File->Upload*, o el botón con forma de flecha a la derecha. Esta instrucción compila el código y lo transmite al microcontrolador. Previamente hay que seleccionar a qué puerto del ordenador está conectada la placa y de qué tipo de placa Arduino se trata. La IDE incluye además algunas otras funciones, algunas de las cuales se van explicando en otros capítulos del proyecto, pero que no son relevantes para dar una visión general de Arduino.

2.3 Shields

Una forma muy habitual de utilizar Arduino es la construcción de *shields* (literalmente "escudos" aunque esta traducción no se usa nunca). Un *shield* consiste en una placa de circuito impreso que tiene unos pines laterales (macho) que se conectan directamente a los pines de Arduino, montando el *shield* sobre la placa. Así un proyecto hecho en la plataforma Arduino suele consistir en uno o varios shields, y un sketch de Arduino que los controle. En este proyecto los prototipos del sintetizador se han construido también en forma de *shields*.

3. DETALLES TÉCNICOS DEL SID

Para comprender el sintetizador construido es necesario conocer las características del chip en que está basado. En este apartado se explica detalladamente, primero ofreciendo una visión de conjunto y luego desarrollando cada parte. Se trata de un sintetizador compatible con la familia de microprocesadores utilizada en los ordenadores Commodore. Estas características están detalladas en la especificación técnica del SID (Commodore semiconductor group, 1986). En la Figura 3.1 se muestra el diagrama de bloques interno del SID. El 6581 utiliza tres voces, encargadas de la generación del sonido. Cada una de ellas está compuesta por tres bloques: el bloque oscilador[1], el generador de envolvente y el modulador de amplitud.

El oscilador controla la frecuencia del sonido producido por la voz, en un rango de 0 Hz a casi 4 kHz. El generador de forma de onda produce cuatro posibles formas de onda a la frecuencia seleccionada: triangular, diente de sierra, pulso rectangular y ruido. Cada forma de onda tiene su propio espectro característico, proporcionando un control sencillo del timbre.

El modulador de amplitud consiste en un amplificador de ganancia variable cuya señal de control es producida por el generador de envolvente, y varía dinámicamente el volumen del sonido producido por el oscilador. Cuando se inicia, normalmente al tocar una nota, el generador de envolvente produce una curva ADSR, que el modulador de amplitud aplica a la señal, y que es programable mediante unos pocos parámetros.

Además de las voces, también tiene un filtro programable. Dicho filtro puede usarse para acentuar, atenuar, o eliminar componentes espectrales de las voces, modificando así su timbre mediante síntesis sustractiva. Incluye también otras funciones, como la posibilidad de utilizar una entrada de audio externa, así como permite al procesador leer algunos registros internos del SID. Éstas no son utilizadas por el sintetizador diseñado, y por eso no se explican en este documento.

[1] Simplificando, el aquí llamado "bloque oscilador" está formado internamente por un oscilador y un generador de forma de onda. En la hoja de características del SID se tiende a llamar oscilador indistintamente al bloque completo y al oscilador en sí, llevando a veces a confusión. Por eso se llamará "oscilador" al oscilador y "bloque oscilador" al conjunto oscilador/generador de forma de onda.

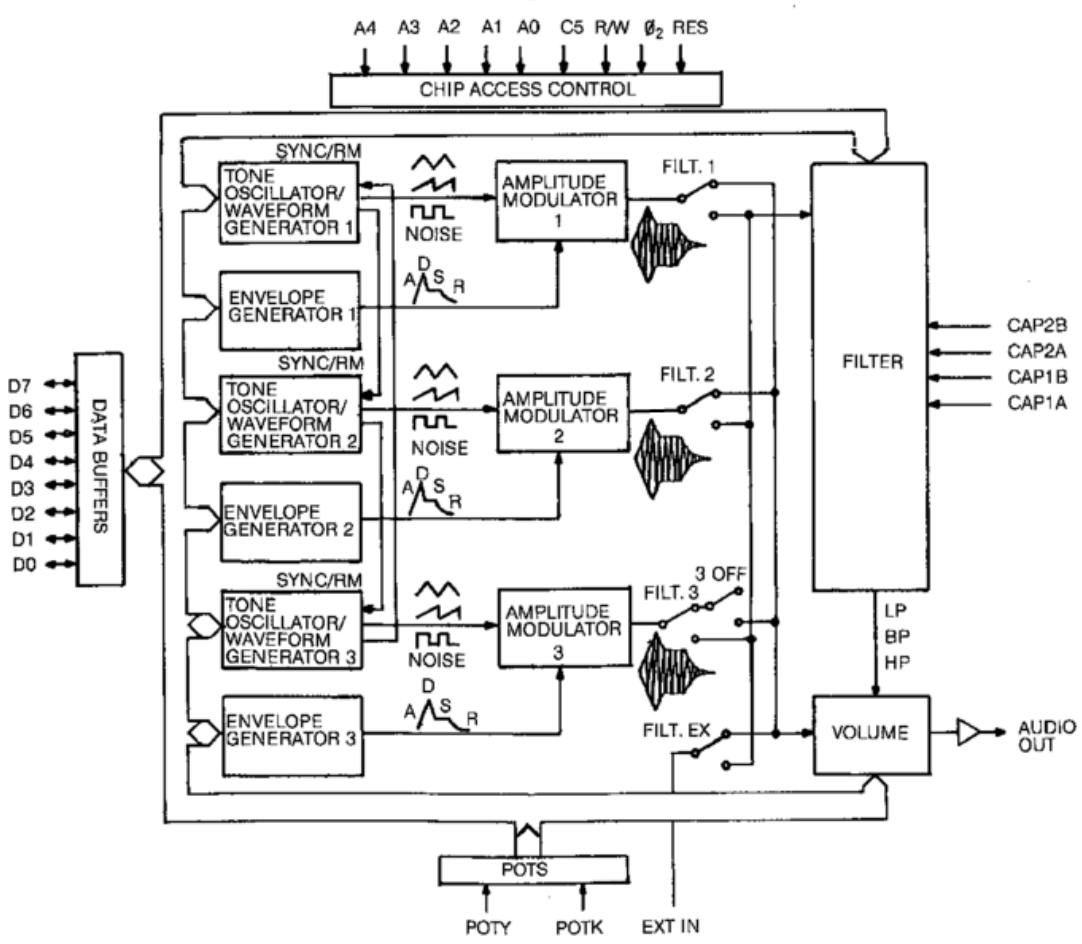

Figura 3.1. Diagrama de bloques del SID

En cuanto a la forma de controlar todos los parámetros, el SID tiene unos registros internos (29 en total) para ello, cada uno de los cuales almacena un valor de 8 bits. En la Figura 3.2 se muestra cómo están distribuidos los registros. La mayoría son de sólo escritura (y son los únicos que se utilizan en este proyecto), y el valor que en ellos se escriba es utilizado para controlar algún aspecto concreto de la generación del sonido. Es importante entender que la única forma de controlar el funcionamiento del SID es escribir valores en los registros, y él los usa internamente para generar el sonido deseado. En otras palabras, y pensando en la Figura 1.1, los registros son equivalentes a las señales de control de los distintos bloques. De hecho, se puede ver que el diagrama del SID es muy parecido al diagrama del sintetizador mostrado en dicha figura, con tres voces.

Por lo tanto escribir valores en los registros es siempre lo que se hace en el sistema en el que funcione el chip (y, por supuesto, eso es lo que hacía el procesador del C64). Cada registro tiene asignada una dirección o número de registro, indicado a la izquierda de la figura, y en el centro está el significado de cada bit de los registros. Algunos de ellos constituyen números, y en otros, como los registros de control, cada bit tiene un significado particular.

	Address					Reg #	Data									
	A4	A3	A2	A1	A0	(Hex)	D7	D6	D5	D4	D3	D2	D1	D0	Reg Name	Reg Type
															VOICE 1	
0	0	0	0	0	0	00	F7	F6	F5	F4	F3	F2	F1	F0	Freq Lo	Write-only
1	0	0	0	0	1	01	F15	F14	F13	F12	F11	F10	F9	F8	Freq Hi	Write-only
2	0	0	0	1	0	02	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0	PW LO	Write-only
3	0	0	0	1	1	03	—	—	—	—	PW11	PW10	PW9	PW8	PW HI	Write-only
4	0	0	1	0	0	04	NOISE	⊓⊓	/|	/\	TEST	RING MOD	SYNC	GATE	Control Reg	Write-only
5	0	0	1	0	1	05	ATK3	ATK2	ATK1	ATK0	DCY3	DCY2	DCY1	DCY0	Attack/Decay	Write-only
6	0	0	1	1	0	06	STN3	STN2	STN1	STN0	RIS3	RIS2	RIS1	RIS0	Sustain/Release	Write-only
															VOICE 2	
7	0	0	1	1	1	07	F7	F6	F5	F4	F3	F2	F1	F0	Freq LO	Write-only
8	0	1	0	0	0	08	F15	F14	F13	F12	F11	F10	F9	F8	Freq Hi	Write-only
9	0	1	0	0	1	09	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0	PW LO	Write-only
10	0	1	0	1	0	0A	—	—	—	—	PW11	PW10	PW9	PW8	PW HI	Write-only
11	0	1	0	1	1	0B	NOISE	⊓⊓	/|	/\	TEST	RING MOD	SYNC	GATE	Control Reg	Write-only
12	0	1	1	0	0	0C	ATK3	ATK2	ATK1	ATK0	DCY3	DCY2	DCY1	DCY0	Attack/Decay	Write-only
13	0	1	1	0	1	0D	STN3	STN2	STN1	STN0	RIS3	RIS2	RIS1	RIS0	Sustain/Release	Write-only
															VOICE 3	
14	0	1	1	1	0	0E	F7	F6	F5	F4	F3	F2	F1	F0	Freq Lo	Write-only
15	0	1	1	1	1	0F	F15	F14	F13	F12	F11	F10	F9	F8	Freq Hi	Write-only
16	1	0	0	0	0	10	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0	PW LO	Write-only
17	1	0	0	0	1	11	—	—	—	—	PW11	PW10	PW9	PW8	PW HI	Write-only
18	1	0	0	1	0	12	NOISE	⊓⊓	/|	/\	TEST	RING MOD	SYNC	GATE	Control Reg	Write-only
19	1	0	0	1	1	13	ATK3	ATK2	ATK1	ATK0	DCY3	DCY2	DCY1	DCY0	Attack/Decay	Write-only
20	1	0	1	0	0	14	STN3	STN2	STN1	STN0	RIS3	RIS2	RIS1	RIS0	Sustain/Release	Write-only
															Filter	
21	1	0	1	0	1	15	—	—	—	—	—	FC2	FC1	FC0	FC LO	Write-only
22	1	0	1	1	0	16	FC10	FC9	FC8	FC7	FC6	FC5	FC4	FC3	FC HI	Write-only
23	1	0	1	1	1	17	RES3	RES2	RES1	RES0	Filt EX	Filt 3	Filt 2	Filt 1	RES/Filt	Write-only
24	1	1	0	0	0	18	3 OFF	HP	BP	LP	VOL3	VOL2	VOL1	VOL0	Mode/Vol	Write-only
															Misc	
25	1	1	0	0	1	19	PX7	PX6	PX5	PX4	PX3	PX2	PX1	PX0	POTX	Read-only
26	1	1	0	1	0	1A	PY7	PY6	PY5	PY4	PY3	PY2	PY1	PY0	POTY	Read-only
27	1	1	0	1	1	1B	07	06	05	04	03	02	01	00	OSC3/Random	Read-only
28	1	1	1	0	0	1C	E7	E6	E5	E4	E3	E2	E1	E0	ENV3	Read-only

Figura 3.2. Mapa de registros del SID

3.1 Las voces

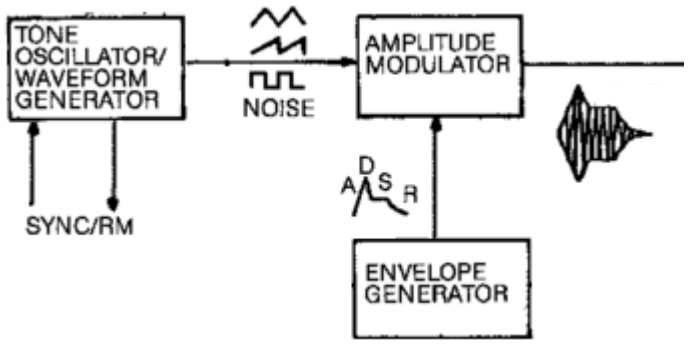

Figura 3.3. Diagrama de bloques de una voz del SID

En la Figura 3.3 se muestra el diagrama de bloques de una voz del SID. Además del esquema básico ya explicado del sintetizador, el SID permite activar dos tipos de modulaciones entre voces: sincronización y modulación en anillo. Se explica a continuación la función de los registros relacionados con una voz (véase la Figura 3.2).

Los dos primeros registros, “Freq Lo” y “Freq Hi”, forman conjuntamente un número de 16 bits (bits bajos y altos, respectivamente) que controla linealmente la frecuencia del oscilador. La frecuencia de salida depende de la señal de reloj que se aplique al chip (en el pin llamado 02), y viene dada por la siguiente expresión:

$$F_{salida} = \frac{F_n \cdot F_{reloj}}{16.777.216} \ (Hz) \tag{3.1}$$

Donde F_n es el número almacenado en los registros de frecuencia. En el *C64* la señal de reloj era de 1 MHz, y esa es también la que se utiliza en el sintetizador construido, por lo tanto, es útil disponer de la fórmula para ese valor de frecuencia de reloj.

$$F_{salida} = F_n \cdot 0{,}0596 \ (Hz) \tag{3.2}$$

De aquí se puede deducir que el rango del oscilador es desde 0 Hz hasta 3,91 kHz. Como F_n es un número de 16 bits esto proporciona un total de 65.536 (2^{16}) posibles valores de la frecuencia en el rango del oscilador, la cual es una resolución suficiente para utilizar cualquier escala, y para hacer transiciones suaves entre nota y nota (*portamento*) sin que los saltos sean perceptibles. Es además útil tener una tabla con los valores de estos registros para cada nota de la escala temperada (la utilizada en la música occidental actual, que divide la octava en 12 semitonos), y el fabricante la facilita en la hoja de características (Commodore semiconductor group, 1986).

Los dos siguientes registros, “PW Hi” y “PW Lo” (*Pulse Width*, anchura del pulso) forman un número de 12 bits que modula la anchura del pulso, y su valor sólo tiene efecto cuando la forma de onda de pulso rectangular está seleccionada. La medida de dicha anchura se hace mediante el ciclo activo (duty cycle), es decir, la relación entre el tiempo en que la señal está a nivel alto y el periodo de la señal, y se suele medir en porcentaje. En la ecuación siguiente se muestra el cálculo de la anchura de pulso según el valor del registro:

$$PW_{out} = \frac{PW_n}{40.95} \ \% \tag{3.3}$$

Donde PW_{out} es la anchura del pulso de la señal generada y PW_n es el número de 12 bits formado conjuntamente por los dos registros. El rango varía entre 0% (cero voltios) para un valor de 0 en los registros y 100% (un nivel de tensión continua) para el valor FFF_H. La resolución (4.092 valores) es suficiente para variar la anchura suavemente sin saltos perceptibles.

El siguiente registro es el registro de control, en el que cada bit tiene su significado particular (véase la Figura 3.2, con el mapa de registros). Controla la forma de onda (bits 7-4), las modulaciones activadas en

la voz ("Ring mod." y "sync", bits 2 y 1 respectivamente), y el momento en que comienzan y terminan las notas ("gate", bit 0). El bit 3 (TEST) no tiene en principio utilidad musical, sino para hacer pruebas.

Los cuatro bits altos del registro de control seleccionan la forma de onda que se tendrá a la salida del bloque oscilador, significando el uno lógico que la forma de onda en cuestión está activada y el cero desactivada. Por ejemplo, para seleccionar la forma de onda triangular en el oscilador 1, habría que escribir en su registro de control (el número cuatro) un valor tal que el bit 4 estuviera a uno (11_H, por ejemplo).

Internamente el bloque oscilador de cada voz tiene un generador independiente para cada forma de onda, y la salida de cada generador aparece en la salida del bloque oscilador si la forma de onda correspondiente está activada en el registro de control. Esto significa que al seleccionar varias formas de onda simultáneamente a la salida se obtendrá una combinación de ellas, más concretamente un AND digital. Esto puede ser peligroso si la forma de onda de ruido está seleccionada, pues el generador de ruido se puede bloquear hasta que se reinicie el chip.

La salida del generador de ruido es una señal aleatoria que varía con la frecuencia del oscilador[1]. Esto permite variar entre ruidos graves, como de vibración, y agudos como de siseos. El generador de ruido es útil para generar efectos de sonido tales como explosiones, disparos, o de percusión (para percusión tiene un sonido muy poco natural, aunque muy característico de la música de los videojuegos de la época).

Los bits 2 y 1 controlan las modulaciones disponibles entre voces: modulación en anillo y sincronización. Cuando se activa una modulación (poniendo a 1 el bit correspondiente) en una voz, en la salida del bloque oscilador se obtendrá la modulación entre dicha voz y el oscilador "modulador", que es siempre el de la voz anterior. Esto se ve más claramente en la Tabla 3-1, o en el diagrama de bloques de la Figura 3.1.

Tabla 3-1. Relación entre las voces para la modulación en anillo y la sincronización

Voz modulada	Oscilador modulador
1	3
2	1
3	2

La modulación en anillo entre dos señales consiste sencillamente en multiplicarlas, pudiendo producir estructuras armónicas (o no armónicas) complejas. Al multiplicar dos tonos puros de frecuencias f1 y f2 (siendo f2 mayor que f1), a la salida se obtienen señales de frecuencias f2+f1 y f2-f1, como se expresa en la ecuación siguiente:

[1] Consiste en un generador de secuencia pseudoaleatoria que toma como señal de reloj uno de los bits intermedios del oscilador, lo que hace que tenga cierta "información tonal" según la frecuencia del oscilador.

$$sen\ (\omega_1 t) \cdot sen(\omega_2 t) = \frac{\cos((\omega_2 - \omega_1)t)}{2} - \frac{\cos\ ((\omega_2 + \omega_1)t)}{2} \tag{3.4}$$

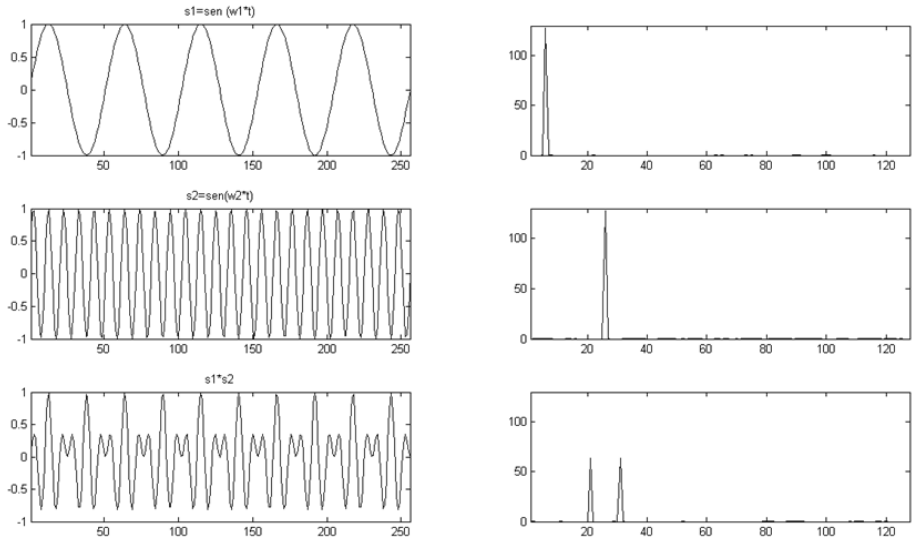

Figura 3.4. Ejemplo de modulación en anillo entre dos tonos puros (5 Hz y 25 Hz). A la izquierda, las señales en el dominio del tiempo; a la derecha, sus respectivos espectros.

En la Figura 3.4 se muestra una modulación en anillo entre dos tonos puros, en el dominio del tiempo y la frecuencia. Todos los espectros están representados en la misma escala, por lo que puede apreciarse claramente la obtención de las frecuencias suma y diferencia. Es interesante observar que si las dos señales están relacionadas armónicamente (es decir, sus frecuencias son múltiplo de la misma frecuencia fundamental, como es el caso del ejemplo) el resultado es armónico también de la misma frecuencia, produciendo un sonido tonal con una estructura armónica muy distinta a la de las entradas. Sin embargo, cuando no estén relacionadas armónicamente, la salida no será armónica, pudiendo tener frecuencias muy dispares. Esto puede ser útil para producir sonidos inarmónicos, similares a campanas, o sencillamente efectos extraños.

Por supuesto, aunque se ha ejemplificado la modulación en anillo con tonos puros para simplificar su comprensión, recuérdese que las señales complejas no están compuestas por un único armónico, sino por varios. En ese caso, las frecuencias suma y diferencia se producen para todos los armónicos de ambas señales, produciendo estructuras armónicas muy complejas.

En el SID, para que sea audible la modulación en anillo tiene que estar seleccionada la forma de onda triangular en la voz modulada, pero no importa la forma de onda en la voz moduladora. En realidad, la modulación en anillo, cuando se activa, sustituye a la forma de onda triangular del bloque oscilador, y por eso tiene que estar seleccionada. La señal que se toma como moduladora es directamente la del oscilador, y por eso no importa la forma de onda.

La otra modulación posible, la sincronización, se trata de un efecto llamado *hard sync*. Esto consiste en reiniciar (poner a cero) el oscilador modulado a la frecuencia del oscilador modulador. En la Figura 3.5 se ilustra un ejemplo, en este caso con una onda triangular. Del oscilador modulador el único parámetro que afecta es su frecuencia, no su forma de onda. Es fácil deducir que la frecuencia fundamental de la señal de salida es siempre la del oscilador modulador, y variando la frecuencia del modulado (preferentemente mayor) respecto al modulador se obtienen variadas formas de onda, con timbres variados.

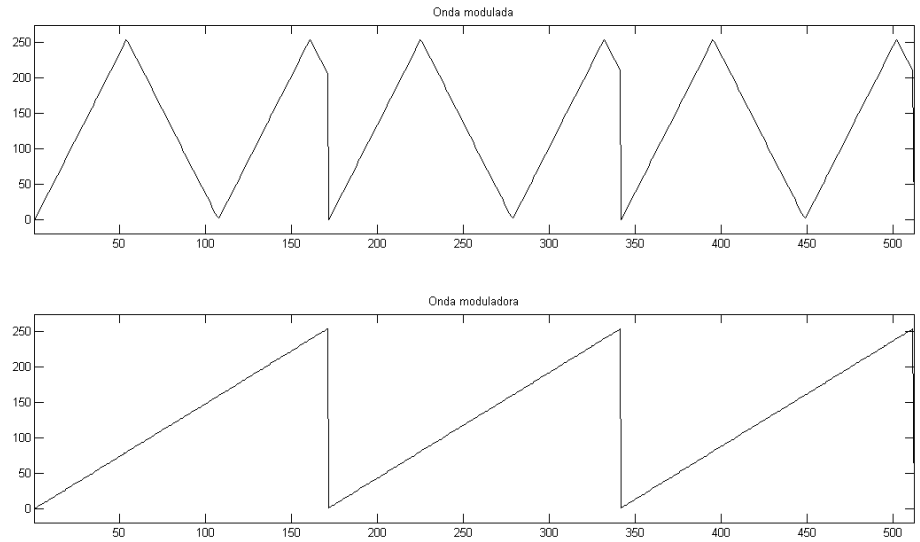

Figura 3.5. Ejemplo del efecto hard sync de una onda triangular

El bit 0 (GATE) del registro de control es el encargado de comenzar y terminar cada nota, controlando el generador de envolvente. El momento en que se escribe a 1 este bit es equivalente al momento en que comienza la nota, y el generador de envolvente comienza la curva ADSR, más concretamente las fases de ataque, caída y sostenimiento. En esta última fase permanece la curva hasta el momento en que este bit se escribe a 0, cuando termina la nota, y comienza la fase de relajación. Como ya se explica anteriormente, con la curva ADSR varía la intensidad de la señal generada por el bloque oscilador.

La curva ADSR es programable mediante los registros “Attack/Decay” y “Sustain/Release”. En el primero están los tiempos de ataque y de caída (cuatro bits altos y cuatro bits bajos, respectivamente), y en el segundo el nivel de sustain y el tiempo de release. Estos parámetros tienen una resolución de 4 bits, lo que proporciona 16 posibles valores a cada uno. El nivel de sustain varía linealmente entre cero y el máximo, y los tiempos de ataque, caída y relajación se dan por la Tabla 3-2:

Tabla 3-2. Valores de los tiempos de ataque, caída y relajación de la envolvente ADSR en el SID

Valor en el registro	Tiempo de ataque	Tiempo de decay/release
0	2 ms	6 ms
1	8 ms	24 ms
2	16 ms	48 ms
3	24 ms	72 ms
4	38 ms	114 ms
5	56 ms	168 ms
6	68 ms	204 ms
7	80 ms	240 ms
8	100 ms	300 ms
9	250 ms	750 ms
10	500 ms	1.5 ms
11	800 ms	2.4 ms
12	1 s	3 s
13	3 s	9 s
14	5 s	15 s
15	8 s	24 s

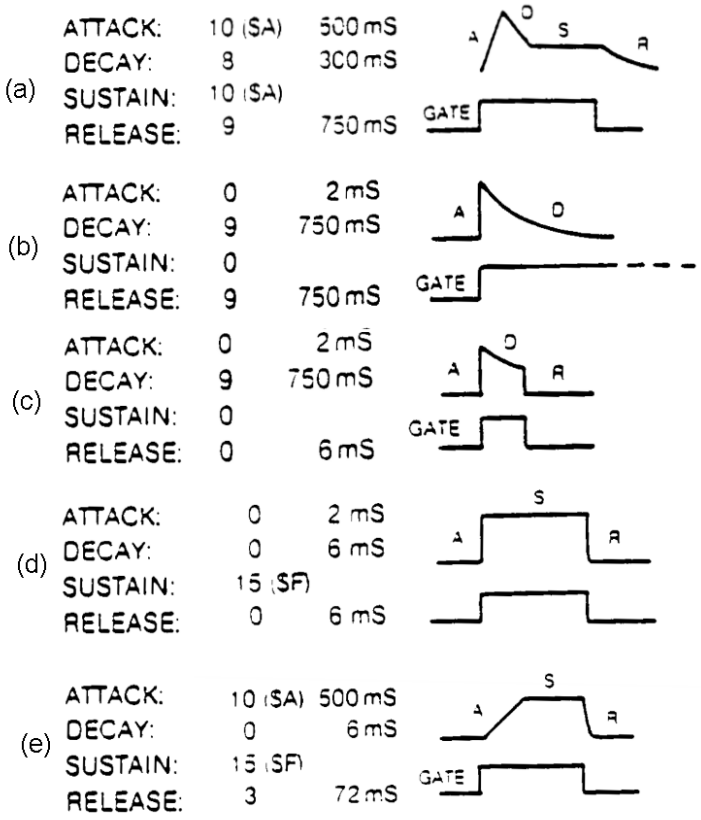

Figura 3.6. Ejemplos de configuraciones de la curva ADSR

En la Figura 3.6 se ven, a modo de ejemplo, algunos posibles usos de la curva ADSR, mediante distintas combinaciones de los parámetros. La curva (a) muestra la evolución natural de los instrumentos de cuerda frotada, o los de viento, usando la curva completa. La (b) correspondería a los instrumentos de percusión. Nótese que, al ser el tiempo de relajación y el de caída iguales, no importa cuándo termina la nota, el sonido cae a su velocidad natural. Caso distinto es el ejemplo (c), que correspondería a instrumentos como el piano, en que al pulsar una nota un martillo golpea una cuerda, proporcionando un ataque rápido y una caída lenta (igual que en los de percusión). Sin embargo, al liberar la tecla, una almohadilla silencia rápidamente la cuerda, efecto que se consigue con un tiempo de relajación más corto. El ejemplo (d), típico del órgano, es el más sencillo: al comenzar la nota, suena al nivel máximo, y al terminar, deja de sonar. Por último, la curva (e) es un ejemplo de que se pueden generar envolventes que no corresponden a instrumentos reales, como la de sonidos reproducidos "hacia atrás" que corresponde a un ataque lento y una caída rápida.

3.2 El filtro

El SID cuenta con un filtro programable, que permite la generación de timbres complejos mediante síntesis sustractiva. El filtro puede funcionar como paso bajo, paso banda o paso alto (o una combinación de las tres). A continuación se explica el funcionamiento del filtro a través de sus registros:

Los registros "FC Lo" y "FC Hi" (registros 21 y 22, respectivamente) forman un número de 11 bits que controla linealmente la frecuencia de corte de los filtros (la misma para paso bajo, paso alto y paso banda, en cuyo caso sería más correcto denominarla frecuencia central). El rango de la frecuencia de corte varía según el valor de los condensadores conectados a los pines "CAP1" y "CAP2" (véase la Figura 3.1, el diagrama del SID), pero el fabricante indica que, para un valor de 1000pF el rango va de 30 Hz y 10 kHz, aproximadamente. De todos modos, también se advierte que el rango puede variar bastante de un chip a otro, por variaciones en el proceso de construcción, o por diferencias en la tensión de alimentación.

El registro 23, "RES/Filt" controla el factor de resonancia de los filtros, y qué voces pasan por el filtro y cuáles no. Los cuatro bits más altos corresponden a la resonancia, es decir, realce (amplificación) a la frecuencia de corte, efecto que se suele medir mediante el factor de calidad[1]. En cuanto a los bits más bajos, en cada uno el valor 1 significa que la voz correspondiente pasa por el filtro, y el valor 0 que va directamente a la salida. El bit 3 ("Filt ex") hace lo propio para una fuente externa de audio, conectada al pin *ext in*.

[1] En la hoja de características del SID no se especifica el rango de valores de este parámetro, únicamente se indica que varía entre la resonancia máxima (F_H) y sin resonancia (0_H).

El registro 24 (*Mode/Vol*) controla el modo en que funciona el filtro (paso bajo, paso banda o paso alto) y el volumen de la mezcla final con los cuatro bits bajos, así como permite desconectar la voz 3 de la salida. Esto último puede ser útil para utilizarla como fuente de modulación sin que aparezca en la salida. Esto se hace escribiendo a 1 el bit 7 (“3 Off”). Los cuatro bits bajos son un número que varía linealmente el volumen de salida, entre cero y el máximo.

Los bits 4, 5 y 6 (“LP”, “BP” y “HP”) activan o desactivan cada uno de los filtros, significando 1 que el filtro en cuestión está activado y 0 que no. En cuanto a la característica de los filtros las pendientes de los filtros paso bajo y paso alto son de 12 dB/octava, y las del paso banda de 6 dB/octava. Cabe destacar pueden seleccionarse varios filtros para conseguir otros efectos. Por ejemplo, al seleccionar los filtros paso bajo y paso alto, se consigue un filtro de ranura, que atenuaría la frecuencia de corte, dejando pasar el resto.

4. DESCRIPCIÓN GENERAL DEL SINTETIZADOR

Las características del sintetizador son:

- Dos modos de funcionamiento: monofónico con tres osciladores y polifónico de tres voces
- Control a través de MIDI: recibe mensajes *Note On* y *Note Off*, para la interpretación musical y *Control Change* para el control de los parámetros del sintetizador. Puede también recibir esta información a través de USB.
- Control grueso y fino de la afinación (*coarse* y *fine*), independiente para cada voz en el modo monofónico y global para las tres para el modo polifónico. El valor de *coarse* transpone el sonido un número entero de semitonos y *fine* cubre valores intermedios en el rango de un semitono.
- Soporte para guardar y cargar configuraciones de parámetros en forma de presets
- Salida de audio con conector jack 3.5 mm
- Alimentación eléctrica por USB (modo normal) o fuente externa de 12V de tensión continua.

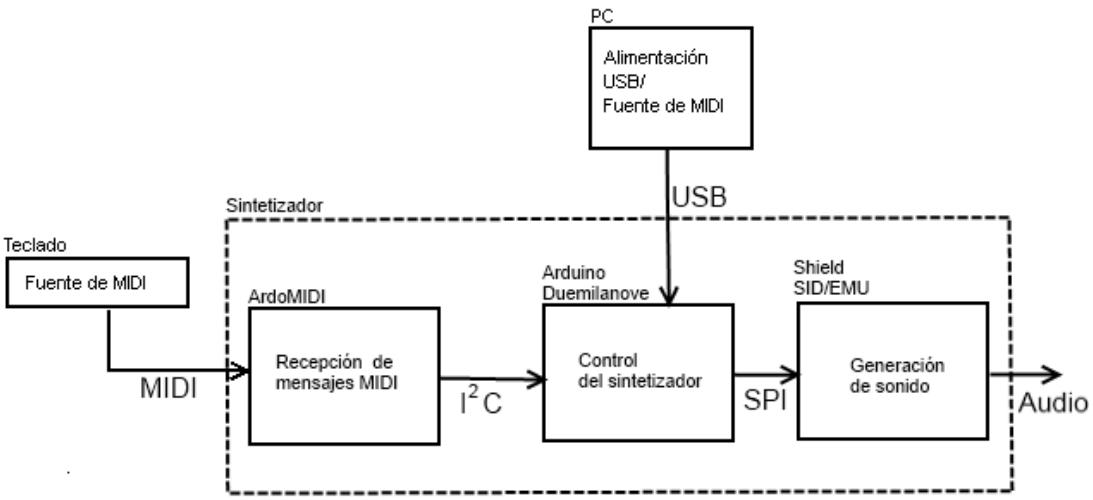

Figura 4.1. Diagrama general del sintetizador

En la Figura 4.1 se muestra el diagrama general del sintetizador, con las conexiones necesarias para su funcionamiento. El bloque de control del sintetizador está hecho en una placa Arduino Duemilanove, con el programa necesario para desempeñar esa función. El bloque receptor de MIDI y el bloque generador de sonido están construidos en placas aparte, en forma de *shields* que se montan sobre la placa Arduino, una encima de otra a modo de "sándwich".

El módulo encargado de la recepción MIDI consiste en un shield llamado ArdoMIDI, diseñado y construido por Raimundo Alfonso, David Rodríguez, Juan Manuel Amuedo y Lino García. ArdoMIDI consiste básicamente en Arduino dotado de interfaz MIDI, en lugar de USB. El programa que tiene cargado tiene la única función de recibir e identificar los mensajes MIDI, y transmitirlos a la placa de control por medio de una conexión I^2C, liberándola a ésta de la tarea de recepción. Este módulo se explica en el capítulo 7.

La placa de control es el "cerebro" del sintetizador. Procesa los mensajes MIDI que recibe de la placa ArdoMIDI y de USB, y en función de ellos y del estado del sintetizador decide qué registros del SID debe escribir (recuérdese que el SID se controla únicamente escribiendo sus registros). Además, genera todas las señales de control que necesita el SID para funcionar (reloj, Chip Select). La conexión con el circuito generador de sonido se hace mediante el protocolo serie SPI (*Serial periferal interface*). El funcionamiento de este módulo se detalla en el capítulo 8.

El bloque generador de sonido recibe la información de la placa de control y genera la salida de audio. Como ya se ha indicado, hay una versión hardware, que utiliza el SID, y una versión software que utiliza un emulador. En la versión "hardware" consiste en básicamente en el SID, con la circuitería necesaria para funcionar con Arduino, y una salida de audio con conector jack de 3,5 mm. En la versión "software", en lugar del SID se trata de un microcontrolador Atmega168 de Atmel (el mismo que llevan las placas Arduino), con un programa emulador del SID. Ambos shields son compatibles con el mismo programa de la placa de control, así que para cambiar sólo hay que sustituir una por otra. El módulo de la versión hardware se explica en el capítulo 5, y el de la versión software en el capítulo 6.

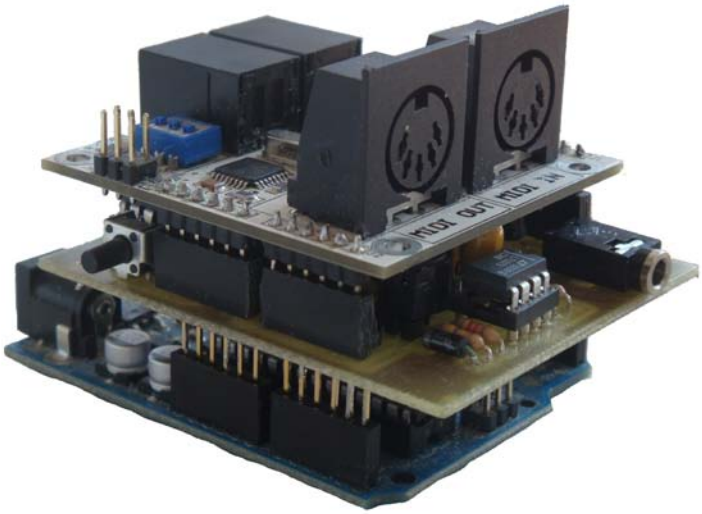

Figura 4.2. Fotografía del sintetizador

Como ya se comentó, el sintetizador se ha construido en forma de shields de Arduino, como se muestra en la fotografía de la Figura 4.2. La placa Arduino, abajo del todo, es la placa de control, sobre la que va montada la placa de generación de sonido, y sobre ella se monta la placa ArdoMIDI. Para ello, la placa de

generación de sonido ha de llevar dos filas de pines laterales, una de pines macho para conectarse a Arduino, y una de pines hembra para conectarse a ArdoMIDI. En la fotografía se muestra la versión hardware del sintetizador, pero la única diferencia entre las dos versiones es la placa de generación de sonido, y se conecta del mismo modo.

Además, aunque no forma parte del sintetizador en sí, es necesario que el ordenador tenga algún software capaz de generar los mensajes MIDI requeridos, y así controlar fácilmene los parámetros del sintetizador. En este caso se ha utilizado un plugin VST llamado *MIDIPads*, que permite generar mensajes MIDI de *Control Change*, programado especialmente para esta aplicación, pero cualquier software capaz de generar estos mensajes sería capaz de realizar esta función. Se explica detalladamente el software adicional utilizado en el capítulo 9.

Todos los ficheros necesarios para hacer funcionar el sintetizador se incluyen en el CD adjunto con el proyecto. Estos ficheros son los diseños de las placas de circuito impreso, lo códigos de la placa de control, la placa de recepción MIDI y el emulador del SID, y todo el software adicional necesario para controlar el sintetizador desde el PC.

5. GENERACIÓN DE SONIDO (HARDWARE)

Esta placa es un shield de Arduino que, al igual que el código de la placa principal, está basada en un sintetizador llamado *SIDaster*, documentado en (Banson). La principal novedad del circuito con respecto al original es la posibilidad de alimentarlo únicamente por USB, mientras que el original requería una alimentación externa de 12 V. Este circuito tiene la función principal de permitir que la placa Arduino controle el SID, escribiendo valores en sus registros por medio de la interfaz SPI (*Serial periferal interface*). En la Figura 5.1 se muestra un diagrama de bloques del circuito de este *shield*, indicando además las conexiones con la placa de control. Al final de este capítulo, en la Figura 5.7 se muestra el esquema completo del circuito.

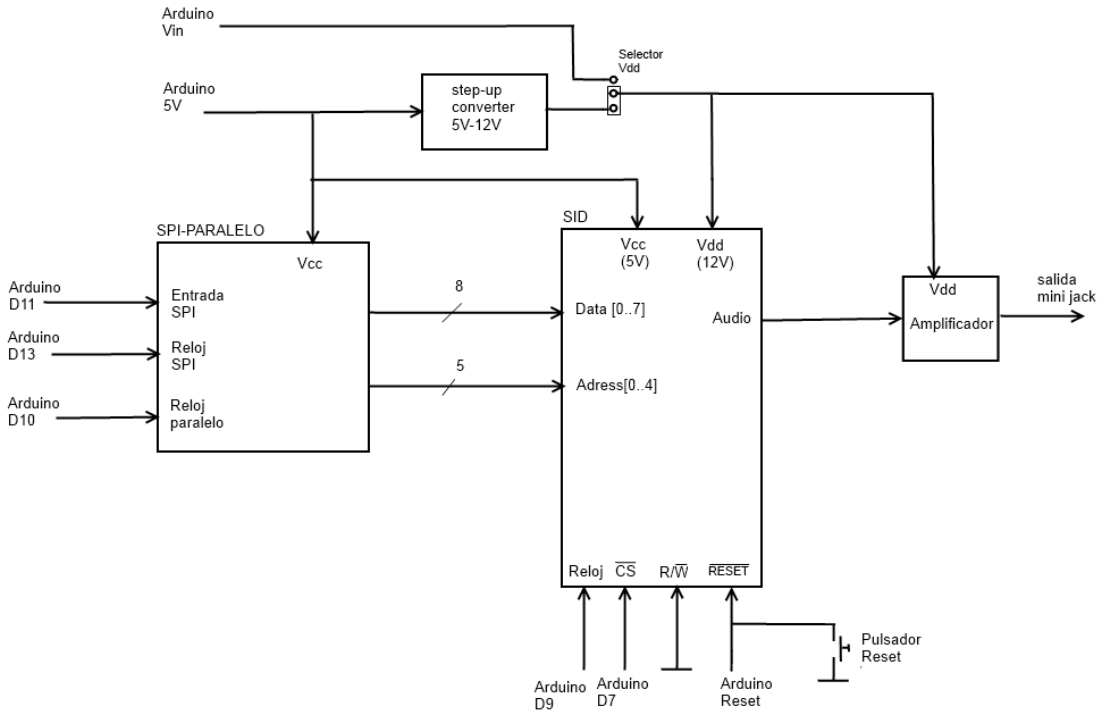

Figura 5.1. Diagrama de bloques del *shield* generador de sonido, versión "*hardware*"

La alimentación del circuito se obtiene del pin "5 V" de la placa Arduino, que, como su propio nombre induce a pensar, proporciona una tensión continua de 5 V, proveniente de la conexión USB. Esta tensión se utiliza directamente para alimentar la circuitería digital. El SID requiere de una fuente de tensión adicional de 12V, y para conseguir esta tensión se ha utilizado un *step-up* converter, que obtiene 12 V a partir de los 5 V proporcionados por la conexión USB. Con los 12 V conseguidos se alimentan tanto el SID como la etapa de salida de audio. También es posible obtener esa tensión adicional del pin "Vin". La placa Arduino Duemilanove permite ser alimentada mediante una fuente de alimentación externa de entre 7 V y 12 V (valores recomendados), de forma alternativa a la conexión USB. La tensión de esa fuente aparece directamente en el pin "*Vin*", y de ahí se puede servir también Arduino para obtener la tensión

habitual de 5V. De esta forma, usando una fuente externa de 12 V de continua, y cambiando el selector Vdd no sería necesario conectar el sintetizador a un ordenador.

Para la escritura de los registros del SID es necesaria una conexión en paralelo, con 5 bits para indicar el número de registro y 8 bits para su valor. Para eso se utiliza un sencillo conversor SPI-paralelo de 16 bits, en que los bits restantes quedan sin conectar. Las señales de control del SID, el reloj de 1 MHz y el *chip select*, son generadas también por la placa de control. La señal *R/*W (lectura/escritura) está conectada directamente a masa (0 V), habilitando siempre la escritura. Se ha incluido también un pulsador que permite conectar el reset de las tres placas a masa, reiniciando el sintetizador completo (también incluido en la versión software).

5.1 Step-up converter

Un *step-up converter* o *boost converter* es un tipo de convertidor conmutado que obtiene a su salida mayor tensión que a su entrada, aunque a costa de ser capaz de proporcionar menos corriente. Está basado en la capacidad de almacenar energía de las bobinas y los condensadores. El principio de funcionamiento se explica utilizando la Figura 5.2. En un primer momento el interruptor S se cierra, y la corriente comienza a fluir a través de la bobina. Durante todo el tiempo que el interruptor está cerrado la bobina se carga con la energía de la fuente. Durante esta fase el diodo está cerrado, pues la tensión en el ánodo es prácticamente cero, y la fuente de tensión de entrada no proporciona energía a la carga, sólo a la bobina.

Pasado un determinado tiempo el interruptor se abre, impidiendo el flujo de corriente a través de él. La bobina tiene una gran cantidad de energía almacenada, y la libera a través del diodo, alimentando la resistencia de carga y cargando el condensador. Una vez más, pasado un tiempo se vuelve a cerrar el interruptor y la bobina vuelve a cargarse, el diodo se corta, y es el condensador quien suministra energía a la carga. El ciclo apertura-cierre del interruptor se repite constantemente (Dekker).

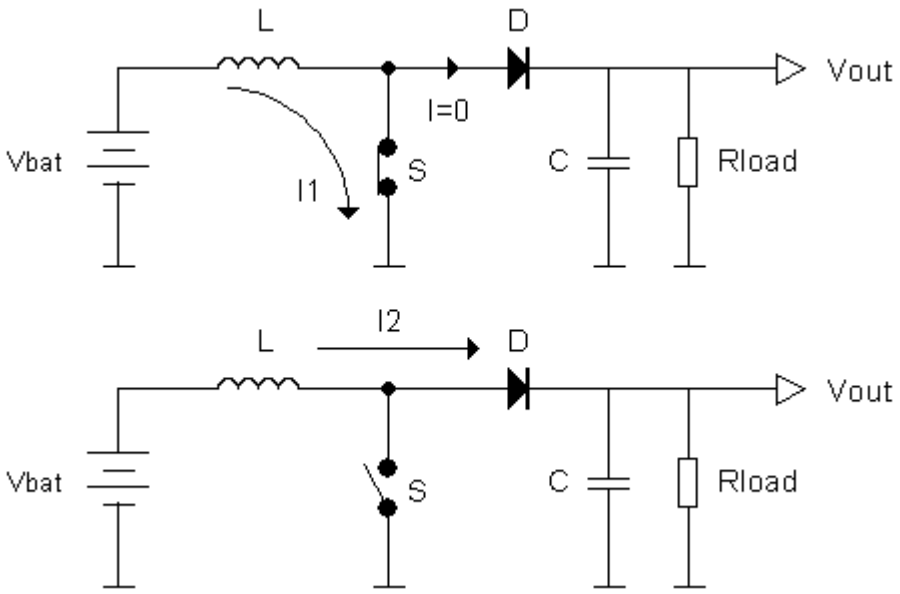

Figura 5.2. Circuito simplificado para el funcionamiento de un step-up converter. Arriba, interruptor cerrado, abajo interruptor abierto.

En los circuitos reales el interruptor es un transistor, y su señal de control (que lo abre y cierra) es una onda de pulso rectangular, con un determinado ciclo de trabajo D:

$$D = \frac{T_{total}}{T_{on}} \tag{5.1}$$

Donde T_{total} es el periodo de la onda y T_{on} es el tiempo en que la onda está a nivel alto, y por lo tanto el interruptor cerrado. Cuanto más tiempo está el transistor cerrado (en relación al tiempo en que está abierto) mayor es la energía que se almacena en la bobina, y mayor también la tensión de salida. Así la tensión de salida es proporcional al ciclo de trabajo D de la señal de control.

Para el convertidor de tensión del sintetizador, se utiliza el chip LT1073, que incluye el interruptor, su señal de control y la forma de regular automáticamente la tensión de salida. En la Figura 5.3 se muestra el circuito del *step-up converter* utilizado. En cuanto al chip, los pines *SW1* y *SW2* son los dos bornes del transistor que actúa de interruptor, y la tensión de entrada se aplica a *VIN*. El pin *Ilim* se usa para limitar la corriente que pasa por el transistor y los pines *SET*, *A0* y *FB* se utilizan para realimentar la salida y poder regular su nivel de tensión automáticamente. Se puede encontrar en (Linear Technology) una descripción detallada de este chip. Para ello se utiliza el divisor de tensión formado por las resistencias *R4* y *R5*, y la tensión de salida Vdd viene dada por la expresión:

$$V_{dd} = (0{,}212) \cdot \left(\frac{R5}{R4} + 1\right) = 12{,}225\ V \tag{5.2}$$

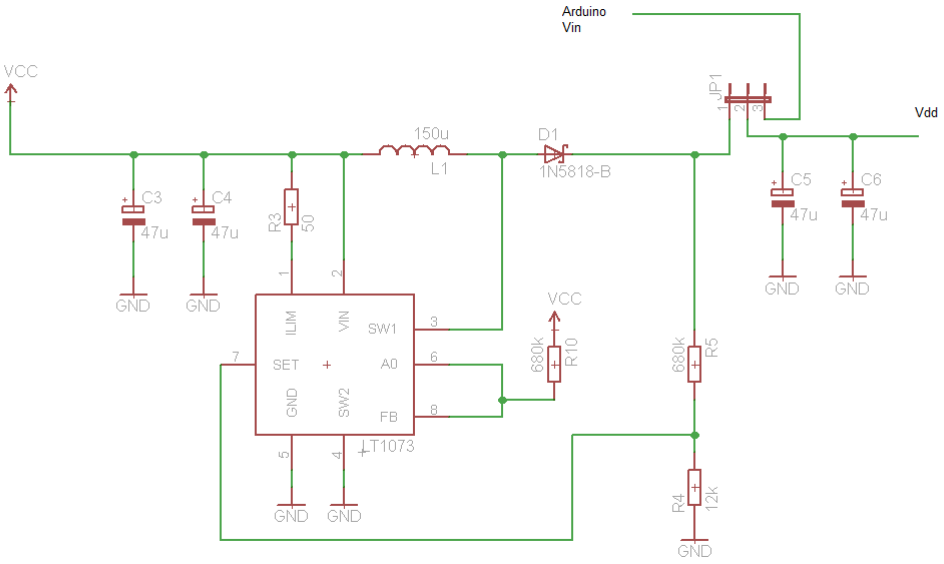

Figura 5.3. Esquema del circuito del step-up converter

Aunque la resistencia de carga no aparece en este circuito, son tanto el SID como el amplificador de salida del sintetizador. Los condensadores *C3* y *C4* actúan como regulador de tensión de la entrada, y los condensadores *C5* y *C6* son equivalentes al condensador de la Figura 5.2. Es típico que las fuentes de tensión conmutadas como ésta obtengan una salida de tensión con rizado, y ese rizado podría aparecer en la salida de audio del SID. Por ello, los condensadores son de tantalio, porque este tipo de condensadores eliminan mejor las altas frecuencias (interesa dejar lo más parecido una tensión continua) y es importante que tensión de alimentación del SID sea lo menos ruidosa posible. Además deben estar lo más cerca posible del pin Vdd del SID.

El selector de Vdd se implementa mediante un jumper (en el esquema *JP1*): tres pines macho y un puente que los puede conectar dos a dos. Conectando los pines 1 y 2 la señal viene del step-up y conectando 2 y 3 la señal viene del pin *Vin* de Arduino.

5.2 Control del SID y conversor SPI-paralelo

Como ya se ha indicado, la placa de control produce todas las señales que necesita el SID para funcionar, y para poder escribir en sus registros. Las señales de control del SID son el reloj, el chip select y R/W. El chip select habilita la lectura o escritura de los registros del SID, cuando está a nivel bajo, es decir, 0V. Esta señal se genera en el pin digital 7 de la placa e control. El reloj es una señal cuadrada con una frecuencia de 1 MHz, que controla todo el funcionamiento del chip, y se genera en el pin 9 de Arduino. La generación de estas señales se explica en el capítulo 8. La señal R/W, cuando está a nivel alto, habilita la lectura de registros del SID y cuando está a nivel bajo habilita la escritura. Como para esta aplicación sólo interesa la escritura de registros, está permanentemente conectada a 0 V.

La escritura de registros del SID se hace por medio de la interfaz SPI (*Serial periferal interface*). SPI es una interfaz que se basa en la transferencia de datos entre registros de desplazamiento (*shift registers*). Un registro es un dispositivo electrónico que es capaz de almacenar una palabra digital, que en este caso es de 8 bits, y tanto su entrada como su salida puede ser serie (bit a bit) y/o paralelo (todos los bits a la vez).

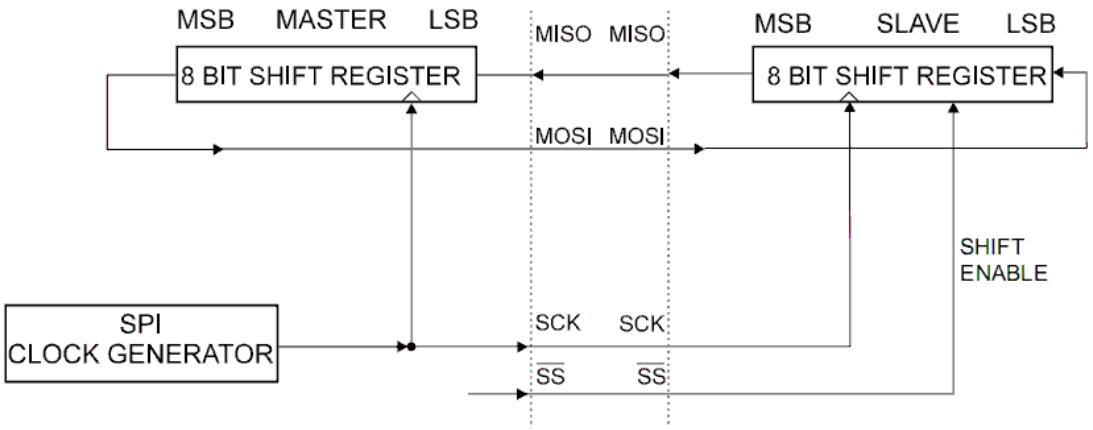

Figura 5.4. Interconexión entre dos dispositivos mediante SPI

En la Figura 5.4 se muestra una interconexión genérica de SPI. Los registros de desplazamiento tienen una entrada de reloj, y en cada ciclo de ese reloj la palabra que contienen se desplaza un bit (en el caso de la figura el desplazamiento es hacia el MSB, aunque este orden se puede invertir), permitiendo la entrada de un nuevo bit. En otras palabras: tienen entrada y salida serie. Esa señal de reloj siempre la genera el maestro, que es quien controla las transferencias. SPI utiliza cuatro líneas, que son:

- *MISO* (*master input slave output*): Transferencia de datos del esclavo al maestro.
- *MOSI* (*master output slave input*): Transferencia de datos del maestro al esclavo.
- *SCK* (*serial clock*): Señal de reloj para los registros de desplazamiento de maestro y esclavo.
- *SS* (*slave select*): Sirve para habilitar (a nivel bajo) el registro de desplazamiento del esclavo.

Viendo la Figura 5.4 es fácil ver que en ocho ciclos de reloj los ocho bits del registro del maestro se vuelcan en el del esclavo a través de la línea *MOSI*, y simultáneamente los ocho bits del registro del esclavo se vuelcan en el del maestro a través de la línea *MISO*. Por supuesto, tanto el maestro como el esclavo pueden escribir o leer sus respectivos registros, aunque no aparezca en la figura. En el sintetizador, el maestro SPI es la placa de control. La velocidad de la transferencia (determinada únicamente por la frecuencia de *SCK*) se ha establecido en 500 kbits/s, y el orden de la transmisión es comenzando por el LSB.

Siendo SPI una interfaz basada en registros, el conversor SPI-paralelo se ha hecho con dos unidades del chip 74HC595, un registro de desplazamiento con entrada serie y salida serie y paralelo, como se muestra en el diagrama de la Figura 5.5. *SER* representa la entrada serie, *QA…QH* son la salida en paralelo, es decir, cada uno de los bits contenidos en el registro, y *QH** la salida serie. *SCK* y *RCK* son el reloj serie y

paralelo, respectivamente. A cada flanco de subida (paso de nivel bajo a alto) de *SCK* los bits del registro se desplazan una posición, y a cada flanco de subida de *RCK* el contenido del registro aparece en la salida paralelo.

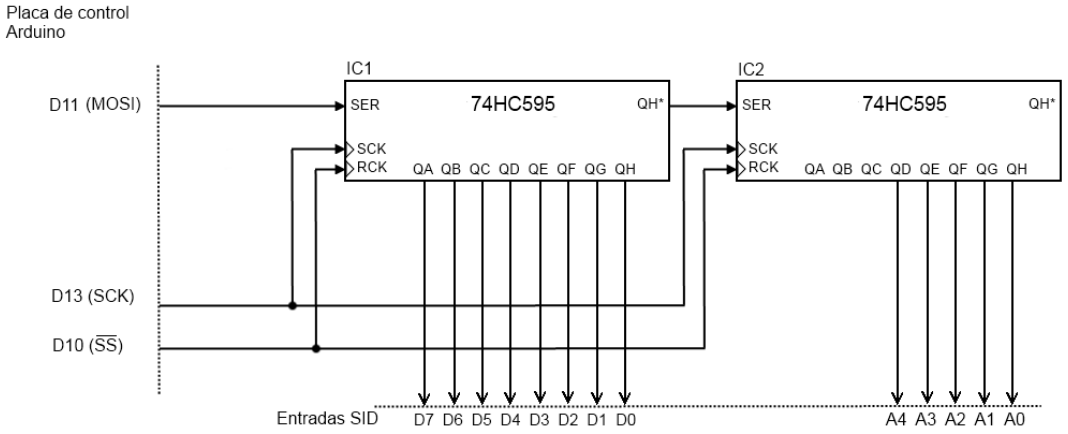

Figura 5.5. Diagrama de bloques del conversor SPI-paralelo

En este caso la línea *MISO* no se utiliza porque sólo se pretende enviar información al SID, y la línea *SS* se utiliza de un modo diferente a como está definido en el protocolo SPI: En este caso el desplazamiento de los registros está siempre habilitado, y *SS* se utiliza como reloj paralelo. Como las direcciones del SID sólo tienen cinco bits, los tres restantes del segundo registro de desplazamiento quedan sin conectar.

El proceso de escritura de un registro en el SID, que se explica también en el capítulo de la placa de control, es:

- La placa de control desactiva el chip select del SID, poniéndolo a nivel alto, inhabilitando la escritura.
- Acto seguido la placa de control envía por SPI primero su dirección y luego su valor (primero el LSB). Justo después de enviar la dirección (ocho bits, los tres MSB a cero), ésta está almacenada en el primer registro de desplazamiento, pero aún no aparece en la salida en paralelo. Conforme va llegando el valor del registro del SID, la dirección va volcándose del primer registro de desplazamiento al segundo.
- Una vez enviados dirección y valor, la placa de control envía un pulso a través de la línea *SS*, haciendo que aparezcan en la salida en paralelo, y por lo tanto en la entrada del SID.
- Por último, cuando ya están la dirección y el valor en la entrada del SID, se activa el chip select (nivel bajo), y el registro correspondiente del SID se actualiza con el nuevo valor. Antes de enviar un nuevo valor la placa de control espera 300 μs, para dar suficiente tiempo al SID para manejar el dato. Este tiempo también será importante para el circuito emulador, para que le dé tiempo a procesar el dato.

5.3 Etapa de salida

La etapa de salida es un circuito con un único transistor, configurado en seguidor de emisor, como se muestra en el esquema de la Figura 5.6. Este tipo de circuito proporciona una ganancia en tensión parecida a la unidad, pero da gran ganancia de corriente, y por lo tanto de potencia, siendo típico de etapas de salida. Este amplificador estaba incluido en el circuito original de *SIDaster*, y, aunque el SID es capaz de dar señal suficiente a un ampificador de potencia o unos auriculares, se ha conservado principalmente para aislar el SID de la salida de audio. Así, el circuito a la salida del SID siempre tiene una impedancia de entrada de 1 kΩ, recomendada por el fabricante, independientemente de qué se conecte a la salida. El SID es un chip escaso, y conectar una impedancia excesivamente baja a la salida podría provocarle daños. La salida de audio del SID está montada sobre una tensión continua de unos 6.5 V, que polariza el transistor. El condensador C7 es necesario para eliminar la componente continua.

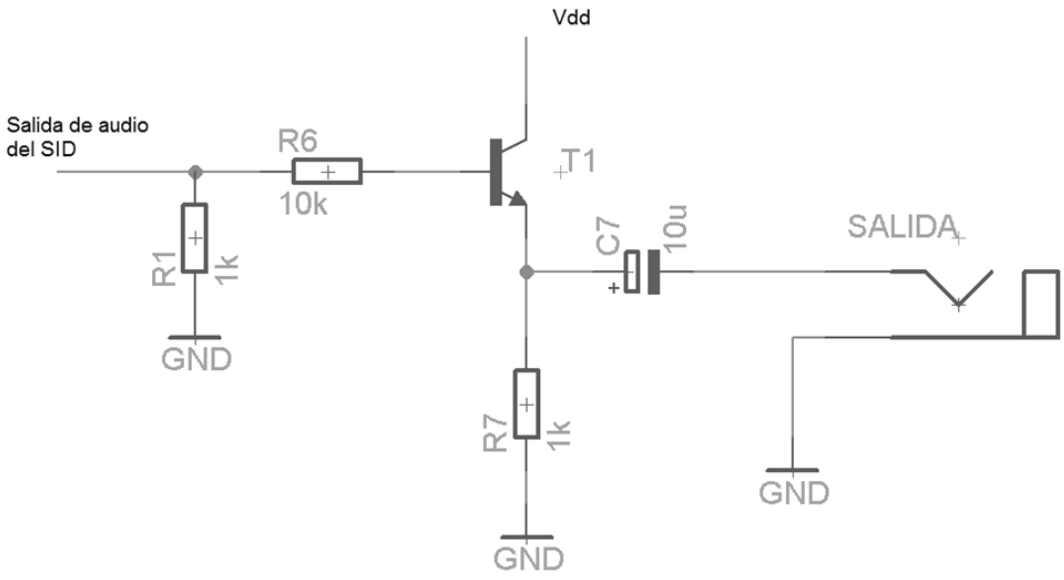

Figura 5.6. Esquema del circuito de la etapa de salida del sintetizador, versión "hardware"

Explicadas todas las partes, el circuito completo se muestra en la Figura 5.7. Los condensadores C8, C9 y C10 son condensadores de desacoplo para la alimentación de los 74HC595 y del SID. Los condensadores C1 y C2 son necesarios para el correcto funcionamiento de los filtros del SID, y su valor recomendado es 1 nF. Con valores más altos se puede obtener mayor control para frecuencias de corte graves (Commodore semiconductor group, 1986). En cuanto a las resistencias R8 y R9, no son propiamente del circuito generador de sonido, sino de la conexión I^2C entre la placa de recepción de MIDI y la de control. Son dos resistencias de pull-up necesarias para una correcta conexión, y se han colocado en este circuito porque ni la placa ArdoMIDI ni Arduino las incluyen. También se incluyen en el circuito de la versión software.

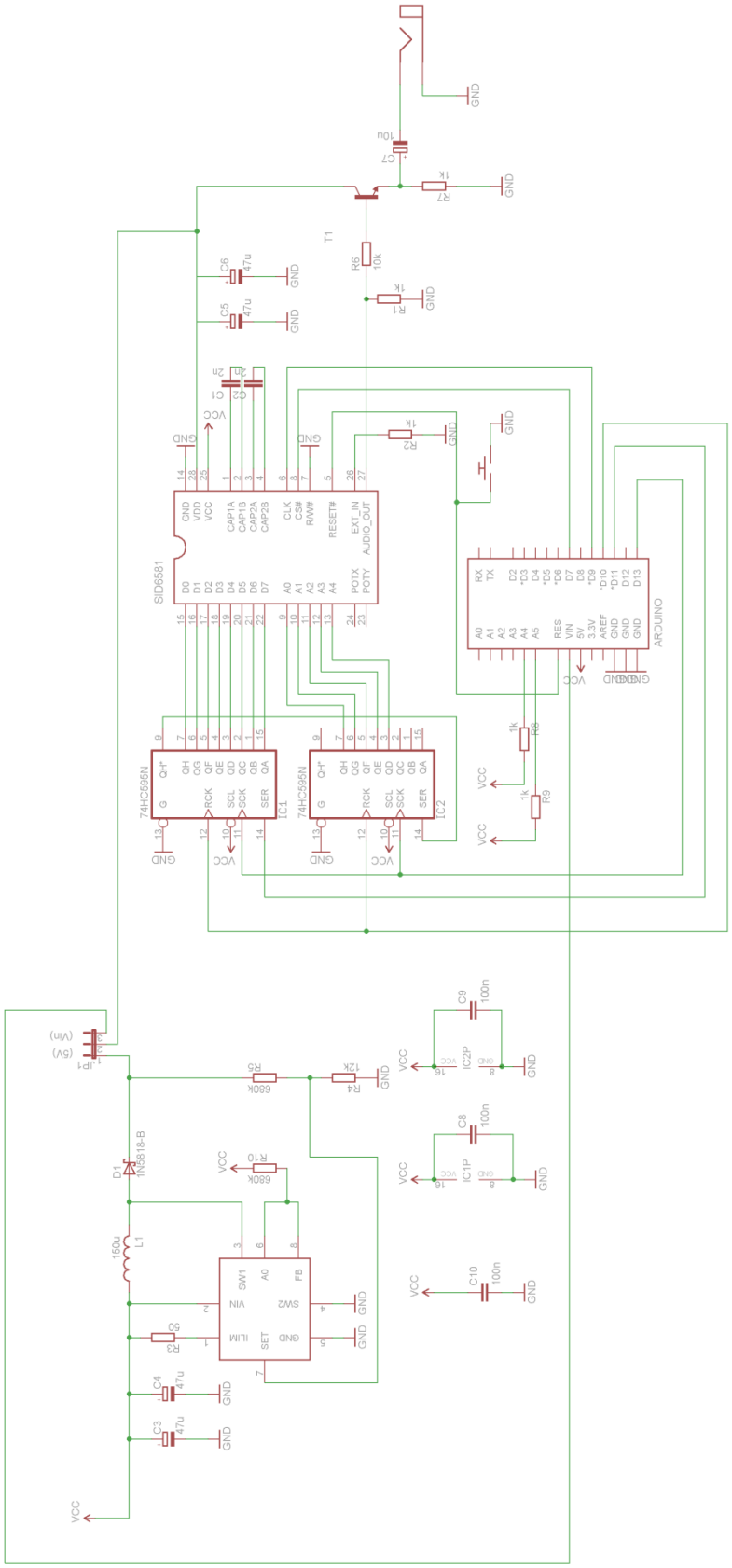

Figura 5.7. Circuito completo del shield generador de sonido, versión hadware

6. GENERACIÓN DE SONIDO (SOFTWARE)

Esta placa está basada en un diseño llamado *SIDuino*, documentado en el sitio web del diseñador (Haberer). Para la generación de sonido este *shield* se sirve un microcontrolador Atmega 168 de Atmel, que es el mismo que utiliza la placa Arduino, con un programa en su memoria que trata de emular el funcionamiento del SID. El código fuente del programa que ejecuta, programado en C[1], está en el fichero *SIDuino_SPI.ino*, incluido en el CD que acompaña al proyecto.

Se han implementado tres voces con formas de onda triangular, diente de sierra, pulso rectangular y ruido, envolvente ADSR y modulación en anillo. No se ha implementado el filtro del SID, y tampoco la modulación *sync* entre voces. El conversor D/A es un modulador PWM (*pulse width modulation*), con frecuencia de muestreo de 31,25 kHz, y profundidad de muestra de 8 bits. Esta función está integrada en el microcontrolador, no haciendo falta circuitería externa.

Como el Atmega 168 se alimenta con 5 V, no es necesaria la conversión a 12 V que se usaba en la versión con el SID. Además, el microcontrolador tiene integrado un módulo de comunicación SPI, por lo que no es necesario conversor alguno. Todo esto hace que el circuito sea mucho más sencillo que la versión hardware, descrita en el capítulo 5. Sin embargo, la complicación se añade en el programa que se ejecuta.

Las principales novedades respecto al diseño original es la recepción de datos vía SPI, para que sea compatible con el programa de la placa de control, y la posibilidad de subir el código desde la IDE de Arduino sin necesidad de desmontar la placa. Además se ha modificado un poco el algoritmo para el cambio de la forma de onda de una voz, y se ha doblado la frecuencia de muestreo. En cuanto al circuito se ha añadido un filtro RC, con frecuencia de corte 15,9 kHz a la salida del conversor D/A. Salvo estas modificiaciones, se ha conservado el programa original.

En cuanto al circuito, que se muestra en la Figura 6.1, entre los pines 9 y 10 del Atmega se conecta un cristal de cuarzo de 16 MHz, que es la frecuencia de reloj utilizada por el microcontrolador. La salida de audio PWM está en el pin 15, y R1 y C1 forman el filtro de salida. La conexión SPI se hace mediante los pines 17, 18 y 19 (MOSI, MISO y SCK, respectivamente). Se incluye la conexión de MISO, que envía datos a la placa de control, aunque no se utiliza, por si fuera necesario monitorizar desde la placa de control alguna variable interna del emulador. El Jumper JP1 es un selector que conecta el pin reset del Atmega o bien al mismo pin de Arduino, para su funcionamiento normal, o bien con el pin 10 de Arduino, para programar el microcontrolador. Se incluyen también las resistencias R3 y R4, que son resistencias de pull-up necesarias para la conexión entre la placa de control y la placa de recepción de MIDI, así como un pulsador que reinicia el sintetizador.

[1] Aunque la extensión habitual de los programas en C es *".c"*, se ha cambiado a *".ino"* para que sea compatible con la IDE de Arduino, facilitando así la programación del microcontrolador. Como el lenguaje de programación de Arduino es compatible con la sintaxis de C, esto no supone ningún problema.

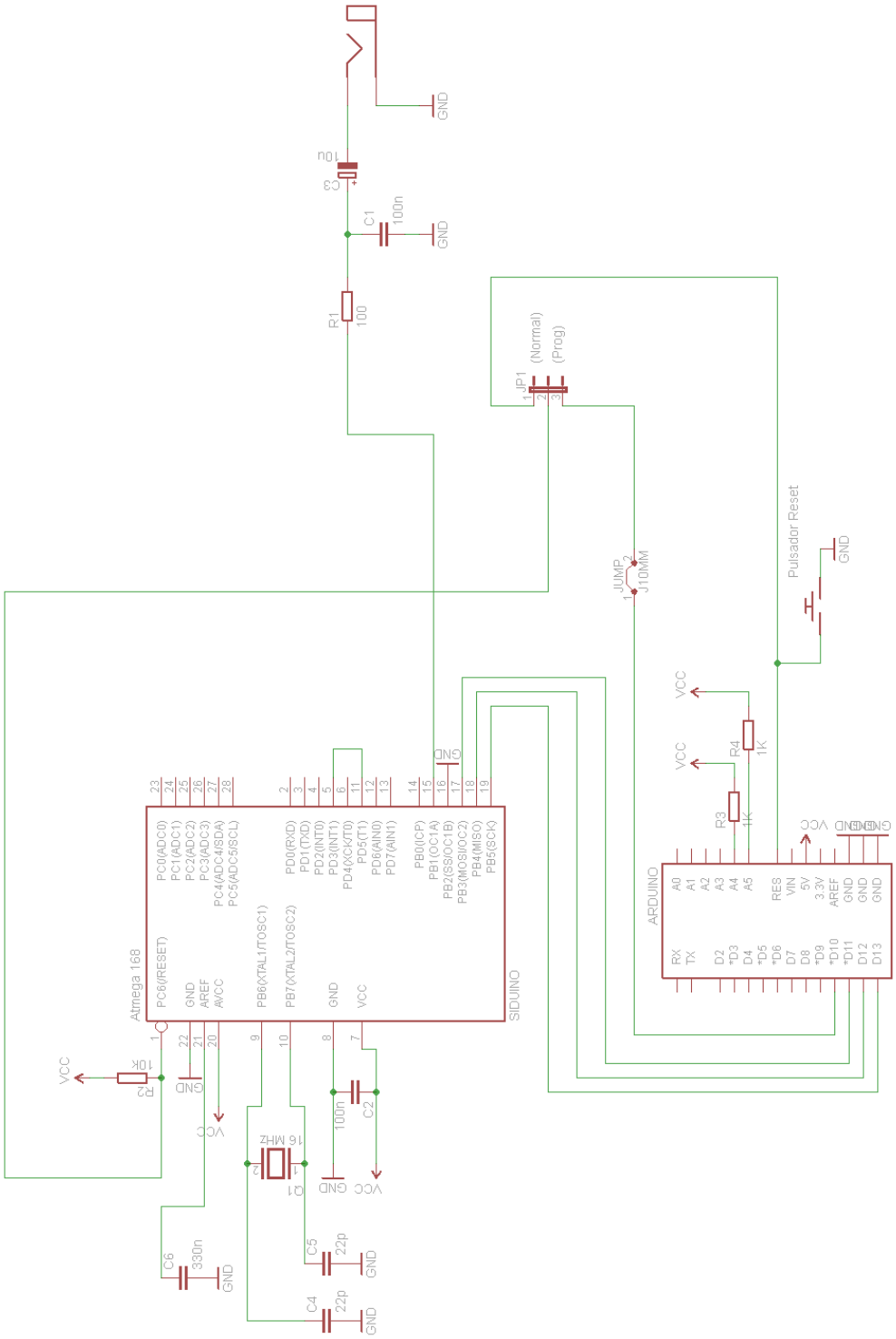

Figura 6.1. Circuito completo del shield de generación de sonido, versión software

6.1 Visión general del programa emulador

Los procesos principales del programa son:

- **Inicialización:** configuración del Atmega para que el resto de procesos puedan realizarse. Sólo se hace una vez, al inicio del programa. Configurar el Atmega consiste en escribir unos valores determinados en unos registros internos que regulan su funcionamiento.
- **Cálculo de la muestra de audio:** se emulan las tres voces del SID, se les aplica la modulación de amplitud (ADSR) y se mezclan para obtener la muestra final de audio. La muestra obtenida se envía al conversor D/A (PWM). Este proceso se realiza cada periodo de muestreo (31,25 kHz).
- **Cálculo de las envolventes ADSR:** este proceso se repite con un periodo de 1 ms. Calcula la amplitud de la señal de cada oscilador en función de los parámetros ADSR y del bit GATE de cada voz. Como los tiempos de la envolvente son mucho más lentos que los de las señales de audio, 1 ms es un periodo suficientemente bajo para muestrear la envolvente.
- **Recepción de datos:** cada vez que el emulador recibe información (escritura de un registro del SID) por SPI, almacena el número de registro y su valor en un buffer FIFO para su posterior procesado.
- **Actualización de los parámetros:** constantemente (esto es, cuando el resto de procesos están inactivos) se comprueba si se han recibido nuevos valores de los registros del SID, se actualizan los valores y según qué registro se haya escrito se modifican las variables correspondientes en las que se basan los procesos que emulan el SID.

En la Figura 6.2 se muestra el diagrama de flujo del programa del emulador. Los procesos de recepción de datos y cálculo de la muestra de audio no aparecen porque no están en el flujo normal del programa, sino que son rutinas de interrupción (ISR, *interrupt service routine*). Una interrupción se produce cuando sucede un determinado evento, interno o externo, por ejemplo que una determinada señal se ponga a nivel alto, o que pase un tiempo determinado. Si la interrupción relacionada con dicho evento está habilitada en el microcontrolador, el procesador deja lo que esté haciendo y pasa a ejecutar la ISR correspondiente (que no es sino una función), y cuando termina se vuelve al flujo normal del programa.

En el caso del emulador del SID, un evento que desencadena una interrupción es la recepción de un byte por SPI (protocolo explicado en el capítulo 6), y en ese momento se ejecuta la ISR de recepción de datos, que almacena los datos recibidos. El otro evento es el *overflow* del contador 1, que se explica más adelante y que sucede exactamente cada 32 μs, desencadenando la ejecución de la ISR de cálculo de la muestra de audio.

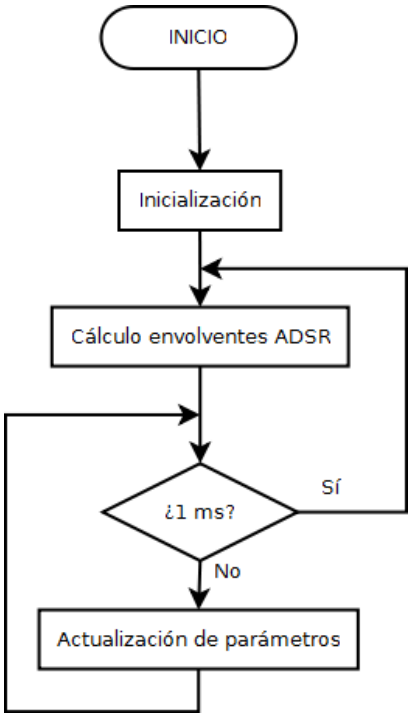

Figura 6.2. Diagrama de flujo del programa principal del emulador

Estos son los procesos principales cuando el emulador ya está funcionando, pero además al final del capítulo se explica cómo hacer funcionar el emulador, es decir, el procedimiento a seguir para introducir el programa en el microcontrolador. A continuación se explica detalladamente el funcionamiento del programa.

6.2 Estructura de registros del SID

Internamente el emulador guarda los valores de cada registro del SID. En el Listado 6-1 se muestra la implementación de esta estructura de registros. Se declara una variable llamada *Sid*, que es una unión del tipo *Soundprocessor*. Las uniones en C permiten almacenar datos de distinto tipo en las mismas direcciones de memoria, en este caso un array de 29 posiciones y una estructura de tipo *Blocks*. En otras palabras, y viendo el Listado 6-1, lo que hace el compilador es lo mismo que haría para declarar únicamente el array: reservar 29 direcciones de memoria (tamaño byte) para su almacenamiento. La diferencia es que permite al programador hacer referencia a esos datos indistintamente como un array o como una estructura de tipo *Blocks*. Por ejemplo, las expresiones *Sid.sidregister[4]* y *Sid.block.voice1.ControlReg* son equivalentes, y direccionan al registro de control de la voz 1. A lo largo de este capítulo, en las secciones de código se incluyen referencias a los registros del SID en ambas formas, por lo que es conveniente tener en mente esta estructura.

```
// SID registers

#define NUMREGISTERS 29

struct Voice            //tipo de estructura que contiene los registros relacionados con una voz
{
        uint16_t    Freq;
        uint16_t    PW;
        uint8_t     ControlReg;
        uint8_t     AttackDecay;
        uint8_t     SustainRelease;
};

struct Blocks           //Tipo de estructura que contiene tres estruct. de tipo voz, y el resto de registros del SID
{
        struct Voice voice1;
        struct Voice voice2;
        struct Voice voice3;
        uint16_t FC;                    // not implemented
        uint8_t RES_Filt;       // partly implemented
        uint8_t Mode_Vol;       // partly implemented
        uint8_t POTX;                   // not implemented
        uint8_t POTY;                   // not implemented
        uint8_t OSC3_Random;// not implemented
        uint8_t ENV3;                   // not implemented
};

union Soundprocessor
{
        struct Blocks block;
        uint8_t sidregister[NUMREGISTERS];
} Sid;
```

Listado 6-1. Implementación de la estructura de registros del SID en el emulador

6.3 Cálculo de la muestra de audio

Este proceso consiste en una ISR que se ejecuta exactamente cada 32 μs (31.25 kHz). En primer lugar se calcula la muestra correspondiente a la salida de cada voz y luego se mezclan las tres muestras obtenidas para obtener la muestra final. El resultado se manda al conversor D/A integrado en el chip, de 8 bits. A continuación se detalla el funcionamiento del proceso, más adelante se explica cómo se consigue que se ejecute cada 32 μs.

La obtención de las muestras de las voces es diferente para las formas de onda “normales” y para la forma de onda de ruido. Cuando la forma de onda seleccionada en una voz no es la de ruido, se utiliza un array en el que se almacena un periodo de la forma de onda. Dicho array se genera en otro proceso (actualización de parámetros), cuando se cambia la forma de onda. La salida se obtiene mediante un puntero que va recorriendo las posiciones del array, “remuestreando” la señal. Este proceso se ilustra en la Figura 6.3, y es similar a la técnica de síntesis por tabla de onda.

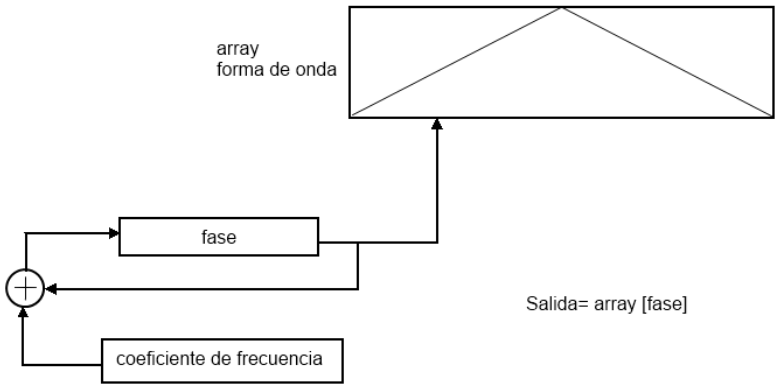

Figura 6.3. Muestreo de la forma de onda de una voz

El recorrido del array se realiza mediante un oscilador acumulador de fase. La fase es el número que se utilizará como puntero al array, y en cada periodo de muestreo se le suma un valor, el coeficiente de frecuencia. Cuando la fase sobrepasa su valor máximo (correspondiente al final del array) vuelve a empezar desde el principio, recorriendo una y otra vez el array. Así, cuanto mayor es el coeficiente de frecuencia, antes se reinicia la fase y mayor es la frecuencia del acumulador, que viene dada por la expresión:

$$F_{out} = F_m \cdot \frac{f_{coef}}{2^N} \tag{6.1}$$

Donde F_{out} es la frecuencia de la señal de salida, F_m la frecuencia de muestreo, f_{coef} el coeficiente de frecuencia y N el número de bits con que está expresada la fase[1]. Así, 2^N es el número de posibles valores que puede obtener la fase. En el emulador, la fase es un número de 16 bits, y la frecuencia de muestreo es 31.25 kHz, y para que la expresión se iguale a la del SID, el coeficiente de frecuencia debe calcularse a partir del valor escrito en los registros del SID:

$$f_{coef} = \frac{1}{2^3} \cdot F_n \tag{6.2}$$

Esta expresión equivale a desplazar tres bits a la derecha el valor del registro. Se utiliza una fase de 16 bits (2^{16} valores), mientras que el array utilizado tiene un tamaño de 256 muestras (2^8), por lo que para recorrerlo hay que despreciar los 8 bits bajos de la fase, desplazándola 8 bits a la derecha.

[1] Los osciladores del SID son en realidad acumuladores de fase de 24 bits y el coeficiente de frecuencia corresponde al valor escrito en los registros. De hecho, esta ecuación es idéntica a la del SID, en el que N=24 y F_m=1 MHz. La salida del oscilador, que tiene forma de diente de sierra, se utiliza en el SID directamente para modelar cada forma de onda, en lugar de direccionar una tabla de onda.

Cuando la modulación en anillo está activada, se hace esto mismo, pero además invirtiendo la señal obtenida si el MSB del oscilador modulador está a 1 (es decir si ha superado la mitad del valor máximo). Esto es equivalente a hacer una modulación en anillo con una señal cuadrada (multiplicar por -1 o por 1), y es parecido a la forma que tiene el SID de hacer la modulación en anillo.

Cuando la forma de onda de ruido está seleccionada, se utiliza un generador de números pseudoaleatorios que se muestra en el Listado 6-2. Este generador de ruido es el mismo para las tres voces, y funciona independientemente de si la forma de onda de ruido está seleccionada en alguna voz o no.

```
/*Variables:
temp1     variable para cálculo
noise     señal de ruido, con precisión de 14 bits
noise8    versión en 8 bits de noise
*/
                    for(k=1;k<2;k++)
                    {
                              temp1 = noise;
                              noise=noise << 1;

                              temp1 ^= noise;
                              if ( ( temp1 & 0x4000 ) == 0x4000 )
                              {
                                        noise |= 1;
                              }
                    }
                    noise8=noise>>6;
```

Listado 6-2. Generador de ruido

Cada periodo de muestreo se actualiza noise8, que corresponde a una señal de ruido muestreada a 31.25 kHz con precisión de 8 bits. Cuando una voz tiene la forma de onda de ruido seleccionada toma la muestra de ruido según un bit intermedio de su oscilador, y el resultado es una señal de ruido que varía con la frecuencia del oscilador.

La muestra obtenida, ya sea por la tabla de onda o por el generador de ruido se multiplica por el valor de la envolvente ADSR en ese momento (calculada por otro proceso, cada 1 ms) y se obtiene la muestra final de cada voz, que posteriormente es mezclada con las demás. El proceso completo de obtención de una muestra de audio de la voz 1 se muestra en el Listado 6-3.

```
/*Variables:
        phase0, phase1, phase2;         fase de osciladores 1, 2 y 3 respectivamente
        sig0, sig1, sig2;               muestra calculada de cada voz.
        tempphase;                      fase temporal para cálculos
        freq_coefficient[3]             array con el coeficiente de frecuencia de cada oscilador
        envelope[3]                     array con el valor actual la envolvente de cada oscilador
        wave0[256], wave1[256], wave2[256]  arrays con un periodo de la forma de onda de los osciladores 1, 2, 3.
*/

tempphase=phase0+(freq_coefficient[0]);     //incremento de la fase
if(Sid.block.voice1.ControlReg&NOISE)       //si en el Reg. de control de la voz 1 está activado el bit NOISE
{
        if((tempphase^phase0)&0x4000)
                sig0=noise8*envelope[0];    //se toma una muestra de ruido, cada cierto tiempo.
}
else                                        //si la forma de onda no es ruido
{
        if(Sid.block.voice1.ControlReg&RINGMOD)     //si está activada la modulación en anillo en esta voz
        {
                if(phase2&0x8000)
                        sig0=envelope[0]*-wave0[phase0>>8];  //invierte la señal con el MSB del osc. 3
                else
                        sig0=envelope[0]*wave0[phase0>>8];
        }
        else
                sig0=envelope[0]*wave0[phase0>>8];
}
phase0=tempphase;
```

Listado 6-3. Obtención de la muestra de audio de la voz 1 en el emulador

En este momento se tienen *sig0*, *sig1*, y *sig2*, que son números de 16 bits con signo que representan la salida de las voces 1, 2 y 3, respectivamente. La mezcla se obtiene simplemente sumando estas tres señales, como se muestra en el Listado 6-4.

```
/*Variables:
        temp    cálculo intermedio de la muestra
        k       muestra de audio final.
*/
temp=0;

 temp+=sig0;
temp+=sig1;
if(!(Sid.block.Mode_Vol&VOICE3OFF))   //sólo mezcla la voz 3 si no está desactivada.
        temp+=sig2;

k=(temp>>8)+128;            //conversión a 8 bits sin signo.

OCR1A=k;                            // Salida a PWM (D/A)
```

Listado 6-4. Obtención de la muestra de audio final y envío a D/A

En resumen: se obtiene una muestra de audio para las tres voces, se multiplican por sus respectivos valores de la envolvente ADSR y se mezclan sumando las tres. La mezcla se obtiene con precisión de 16 bits con signo, y se convierte a 8 bits sin signo, que es la resolución del conversor D/A.

La forma de onda de cada voz contenida en el array es un número de 8 bits con signo, lo cual significa que está comprendida entre -128 y 127. La envolvente está calculada con precisión de 8 bits, y su valor varía entre cero y un valor máximo. El valor máximo de la muestra final de audio es el obtenido si coincide en las tres voces el valor máximo de la forma de onda (127) con el valor máximo de la envolvente, y el valor mínimo si coincide en las tres el valor mínimo de la forma de onda (-128) con el valor máximo de la envolvente, y son:

$$\text{(a)} \quad MAX_{muestra} = MAX_{envolvente} \cdot 127 \cdot 3 \tag{6.3}$$

$$\text{(b)} \quad MIN_{muestra} = MAX_{envolvente} \cdot (-128) \cdot 3$$

El rango de un número de 16 bits con signo varía entre -32.768 y 32.767, y tanto el máximo como el mínimo deben estar por debajo de esos valores, para evitar la saturación de la mezcla. Como valor máximo de la envolvente se ha utilizado 74, lo cual da un rango a la señal final de audio como se muestra en la Tabla 6-1, dentro de los límites del conversor.

Tabla 6-1. Rango de la muestra de audio obtenida por el emulador

	16 bits con signo	8 bits con signo (>>8)	8 bits sin signo (+128)
$MAX_{muestra}$	28.194	110	238
$MIN_{muestra}$	-28.416	-111	17

La muestra final obtenida en la mezcla se envía al conversor D/A. Como ya se ha dicho, la conversión D/A se hace por PWM. En este proceso, la salida es una señal de pulso rectangular, cuyo ciclo activo es proporcional a la entrada digital. Como el valor medio de una señal de este tipo es proporcional al ciclo activo, también es proporcional a la entrada digital, y la señal analógica final se debe obtener mediante un filtrado paso bajo.

Para la modulación PWM se utilizan los módulos contadores del Atmega. En concreto se utiliza el módulo *Timer/Counter 1*, configurado en modo "*Fast PWM, 8 bits*". Este contador se usa tanto para la conversión PWM como para temporizar la ISR de cálculo de la muestra de audio. El funcionamiento del contador en este modo de PWM se puede entender viendo la Figura 6.4.

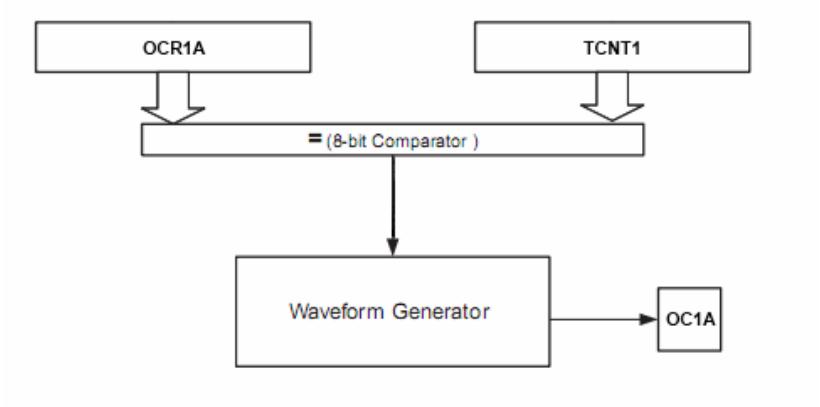

Figura 6.4. Esquema simplificado de módulo contador funcionando como PWM

El contador consta de un registro de 8 bits (TCNT1) que a cada ciclo de reloj se incrementa en 1, hasta llegar al máximo, 255 (8 bits) y vuelve a empezar desde 0. Si se dibujara el valor de este registro en función del tiempo, tendría forma de diente de sierra. La fuente de reloj del módulo contador está programada para ser de 8 MHz. La frecuencia de reinicio del contador (F_{PWM}) es, por lo tanto:

$$F_{PWM} = \frac{F_{clk}}{2^8} = 31.25\ (kHz) \tag{6.4}$$

Constantemente se compara el valor de este registro con el registro del comparador, OCR1A, que es la entrada del modulador PWM. El generador de forma de onda obtiene la salida (en el pin OC1A) en función de la salida del comparador. En este caso está programado para obtener 5 V si OCR1A es mayor que TCNT1 y 0V si OCR1A es menor que TCNT1.

Además, cada vez que el contador llega al nivel máximo (32 μs), se produce una interrupción por overflow del contador 1, y el procesador pasa a ejecutar la ISR que engloba todo el proceso de cálculo recién descrito. Esa es la forma de conseguir que se ejecute la función a la frecuencia deseada. En la Figura 6.5 se muestra el cronograma del *Timer/Counter 1* en el emulador, incluyendo los momentos en que se ejecuta la ISR de cálculo de la muestra de audio. La conversión PWM puede funcionar muy bien siempre que la frecuencia del pulso sea bastante mayor que la frecuencia de muestreo, pero en este caso, por problemas de tiempo de procesado, es la misma, proporcionando una calidad de sonido bastante inferior a la del SID.

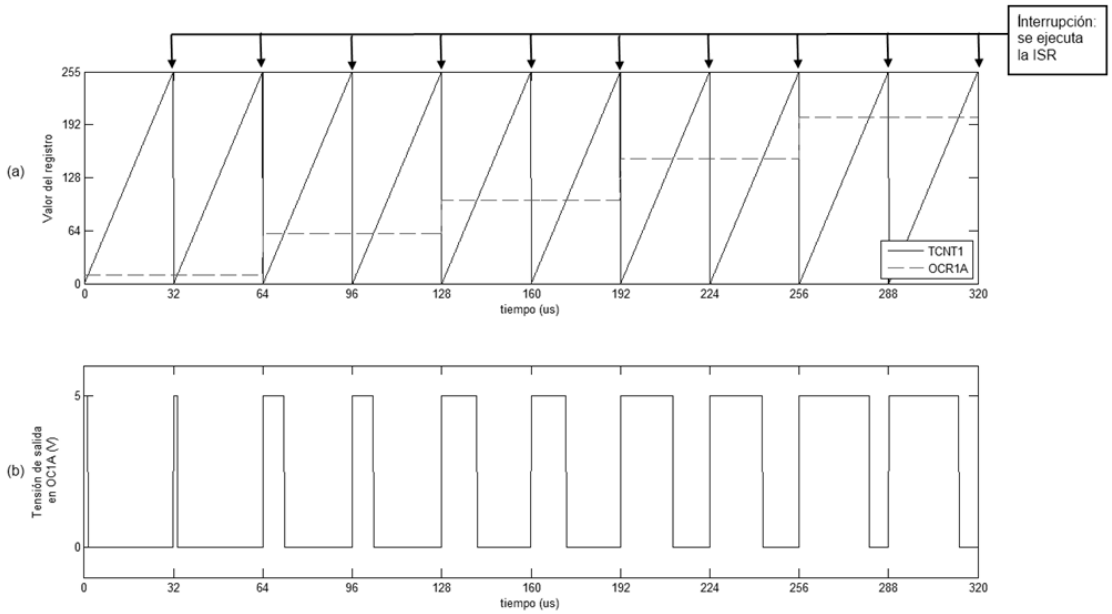

Figura 6.5. Cronograma del Timer Counter 1 en modo Fast PWM, 8 bits. a) Valor de los registros TCNT1 y OCR1A b) Tensión de salida

Aunque no forma parte del proceso de cálculo de la muestra de audio, esta misma rutina se utiliza para incrementar el contador de milisegundos, utilizado para temporizar el cálculo de la envolvente ADSR. Se actualiza el contador de milisegundos cada 31 periodos de muestreo, el equivalente a 0.992 ms (para conseguir exactamente 1 ms habría que incrementarlo cada 31.25 periodos, lo cual no es posible, pero es una aproximación suficientemente buena).

```
/*Variables:
        timer       contador  de periodos de muestreo
        mstimer     contador de milisegundos. (variable global).
*/
timer--;
if(timer==0)
{
        mstimer++;
        timer=31;               // mstimer rate=0.992 ms
}
```

Listado 6-5. Contador de milisegundos

Una última consideración: hasta ahora se ha dado por supuesto que la fuente de reloj del contador era de 8 MHz, pero no se ha explicado la obtención de dicha señal. La fuente de reloj del contador es configurable, y puede ser un divisor (1, 8, 64, 256, 1024) del reloj del sistema, 16 MHz, o una fuente de reloj externa. En este caso se necesitan 8 MHz, pero esa frecuencia no es accesible directamente desde el reloj del sistema. Por ello lo que se hace es configurar el reloj del contador como una fuente externa y usar el *Timer/Counter 2* para generar la señal de 8 MHz.

Con la configuración del reloj se pretende obtener una señal cuadrada de 8 MHz. Para ello se utiliza el *Timer/Counter 2* del Atmega, configurado en modo *CTC* (*Clear timer on compare match)*: Esto significa que el contador (TCNT2) se incrementa hasta que su valor coincide con uno dado (concretamente en el registro OCR2B), y vuelve a empezar. Además el generador forma de onda se configura para conmutar (pasar a 5 V si está en 0 V y viceversa) cada vez que conmuta (modo *toggle on compare match*). Si el valor del registro OCR2B tiene el valor cero, se produce coincidencia en cada ciclo de reloj, y el resultado es una señal de mitad de frecuencia que la de reloj, es decir, 8 Mhz. Por esto en el circuito están conectados los pines 5 (OC2B, salida del *Timer/Counter 2*) y 11 (T1, fuente de reloj del *Timer/Counter 1)*.

6.4 Cálculo de la envolvente ADSR

En este proceso se calcula el valor de la envolvente en cada momento, no los parámetros ADSR. Los valores obtenidos se almacenan en el array *envelope [3]*, que es el que posteriormente se multiplica por la muestra de audio de cada voz, modulando su amplitud. Para determinar el momento de la curva en que se encuentra el generador de envolvente de una voz, se utilizan los siguientes estados:

Tabla 6-2. Fases del generador de envolvente

	GATE	Attackdecay_flag	Nivel de la envolvente
Attack	1	TRUE	menor que el máximo
Decay	1	FALSE	mayor que el de sustain
Sustain	1	FALSE	Igual que el de sustain
Release	0	TRUE	Mayor que cero

Donde *GATE* es el bit 0 del registro de control de la voz, y *attackdecay_flag* es una bandera que indica si está en la fase de ataque o en la de caída. En el proceso de cálculo de la envolvente, según el momento de la curva en que se esté:

- Durante la fase de ataque a la envolvente se le suma un incremento fijo en cada ciclo de 1 ms, y cuando llega al nivel máximo se cambia *attakdecay_flag* a FALSE, para indicar que comienza la fase de caída.
- Durante la fase de caída (y sostenimiento) a la envolvente se le resta un incremento fijo en cada ciclo de 1 ms, hasta llegar al nivel de sostenimiento, en el que permanece hasta que el bit GATE del registro de control se pone a 0.
- En ese momento se vuelve a activar la bandera *attackdecay_flag* para que la próxima vez que se active GATE entre de nuevo en la fase de ataque, y se decrementa linealmente hasta llegar a 0.

Este proceso se repite para las tres voces, y se muestra en el Listado 6-6.

El valor de cada incremento, así como el nivel de sustain no se calcula en este proceso, sino en el proceso de actualización de parámetros, cada vez que modifica un valor de la curva ADSR. En este proceso sólo se utilizan los valores ya calculados. Se habrá observado que en el emulador todos los tramos de la curva

son lineales, es decir, corresponden a una recta (el incremento es fijo). Normalmente en aplicaciones de audio es deseable que los valores de volumen usen curvas exponenciales, para aproximarse más a la percepción de la intensidad, pero esto implicaría almacenar las curvas en memoria, y no queda espacio suficiente en el emulador. Cabe mencionar que en el SID el ataque es lineal, mientras que la caída y la relajación sí son exponenciales (Varga, 1996).

```
/*Variables:
        Sid.sidregister[29]     array que contiene los registros del SID
        controll_regadr[3]      array que contiene la dirección del registro de control de cada voz
        attackdecay_flag[3]     array con la bandera attackdecay de cada voz
        amp[3]                  array con el valor de la envolvente de cada voz, con precisión de 16 bits.
        envelope[3]             valor de amp [n] convertido a 8bits. Es el que se usa para el cálculo de la muestra de audio.
        m_attack[3]             array con el incremento de la envolvente de cada voz para la fase de ataque
        m_decay[3]              array con el decremento de la envolvente de cada voz para la fase de caída
        m_release[3]            array con el decremento de la envolvente de cada voz para la fase de relajación
        sustain_level[3]        array con el nivel de la envolvente de cada voz para la fase de sustain.
Constantes
        MAXLEVEL        19000   valor máximo de amp[3], con 16 bits. Al convertir a 8 bits se convierte en 74
*/
for(n=0;n<OSCILLATORS;n++)              //OSCILLATORS =3
{
        if(Sid.sidregister[controll_regadr[n]]&GATE)            //GATE activado: attack,decay,sustain
        {
                if(attackdecay_flag[n])                         // fase attack
                {
                        amp[n]+=m_attack[n];
                        if(amp[n]>MAXLEVEL)
                        {
                                amp[n]=MAXLEVEL;
                                attackdecay_flag[n]=FALSE; //si se alcanza el máximo, cambiar a decay/sustain
                        }
                }
                else                                            // fase decay/sustain
                {
                        if(amp[n]>level_sustain[n])
                        {
                                amp[n]-=m_decay[n];
                                if (amp[n]<level_sustain[n])  { amp[n]=level_sustain[n];}
                        }
                }
        }
        else                                            // GATE desactivado: release
        {
                attackdecay_flag[n]=TRUE;
                if(amp[n]>0)
                {
                        amp[n]-=m_release[n];
                        if(amp[n]<0)
                                amp[n]=0;
                }
        }
        envelope[n]=amp[n]>>8;          //conversión a 8 bits  }
```

Listado 6-6. Cálculo de las envolventes de las voces

6.5 Recepción de información

Para el almacenamiento de la información se ha implementado un buffer FIFO (*first in, first out*), donde se almacenan los bytes recibidos por SPI. Se puede ver un esquema del buffer en la Figura 6.6. El buffer consiste en un array de tamaño *SIZE*, y dos punteros que direccionan sus datos: la cabeza (*head*) apunta a la dirección en que se introducirá el siguiente byte, y la cola (*tail*) direcciona el próximo byte que se extraerá. El buffer de recepción implementado tiene un tamaño de 64 bytes (32 pares dirección-valor de registro del SID).

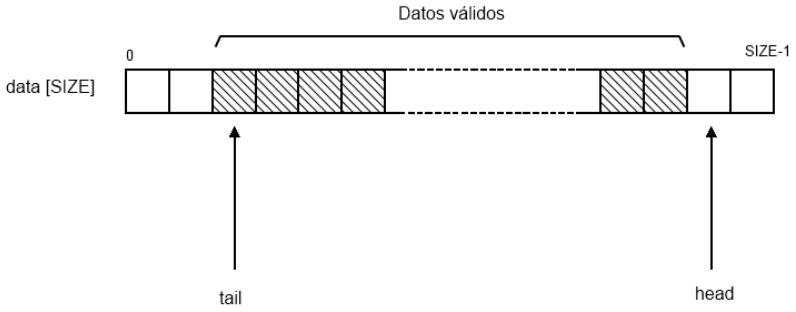

Figura 6.6. Esquema del buffer FIFO

El recorrido del array es circular, es decir, cuando uno de los punteros excede el tamaño del buffer vuelve a las primeras posiciones (por ello, este tipo de buffer se llama "en anillo"). Cuando, al introducir un dato, la cabeza alcanza a la cola, se entiende que el buffer está lleno, y no se pueden introducir más bytes. Por el contrario, cuando al extraer bytes la cola iguala a la cabeza el buffer se ha vaciado y ya no hay datos válidos en él. En ambos casos la cabeza y la cola del buffer apuntan a la misma posición, por lo que para distinguir ambos estados hace falta una bandera adicional que se ha llamado *notfull*. En la Tabla 6-3 se muestran los posibles estados del buffer, con los valores de las variables correspondientes.

Tabla 6-3. Estados del buffer FIFO

	head, tail	***notfull***	**Acciones**
Vacío	*head* = *tail*	TRUE	Se pueden introducir bytes, pero no se pueden extraer.
Lleno	*head* = *tail*	FALSE	Se pueden extraer bytes, pero no se pueden introducir.
Ni vacío ni lleno	*head* ≠ *tail*	TRUE	Se pueden introducir y extraer bytes.

El buffer, al igual que prácticamente todo lo demás se ha implementado mediante variables globales, como se muestra en el Listado 6-7.

```
#define     BUFFER_SIZE         64          //El tamaño del buffer.
#define     BUFFER_IS_EMPTY             ((SPI_buffer_head==SPI_buffer_tail) && (SPI_buffer_notfull))  // TRUE si vacío, FALSE si no.

uint8_t SPI_buffer_data[BUFFER_SIZE];
int8_t SPI_buffer_head=0;
int8_t SPI_buffer_tail=0;
int8_t SPI_buffer_notfull=1;//inicialmente vacío
```

Listado 6-7. Declaración de las variables del buffer del emulador

En este caso los bytes siempre se reciben de dos en dos (primero la dirección del registro, luego su valor) a través de la interfaz SPI. Cuando se recibe un byte completo a través de esta interfaz ocurre una interrupción, y el procesador pasa a ejecutar la rutina de interrupción que almacena los datos, y se muestra en el Listado 6-8. SPDR (*SPI data register)* hace referencia al registro de datos del módulo SPI, que es donde se almacena el dato recibido. Por lo tanto, en la primera línea lo que se hace es tomar el dato recibido, que está en dicho registro. La segunda línea espera a que se reciba un segundo byte. SPSR (*SPI Status Register*) es un registro que contiene, entre otras, una bandera llamada SPIF (*SPI Interrupt flag)* que se activa cuando se recibe un byte, y que se desactiva automáticamente al leer el registro SPDR. Por lo tanto esperar a que esa bandera se active equivale a esperar a recibir el segundo byte.

```
ISR (SPI_STC_vect)              //sintaxis para declarar la ISR para recepción SPI
{
        if (SPI_buffer_notfull)
        {
                SPI_buffer_data[SPI_buffer_head]=SPDR;          //almacena el primer byte
                loop_until_bit_is_set(SPSR, SPIF);              //espera a que se haya recibido el segundo byte.
                SPI_buffer_data[SPI_buffer_head+1]=SPDR;        //almacena el segundo byte en la siguiente posición

                SPI_buffer_head = (SPI_buffer_head +2) % (BUFFER_SIZE);   //incremento circular, dos bytes
                SPI_buffer_notfull = (SPI_buffer_head - SPI_buffer_tail);  //si se ha llenado el buffer not_full=0;
        }
}
```

Listado 6-8. Rutina de recepción de datos del emulador

En cuanto a la extracción de datos del buffer, el proceso es parecido: si el buffer no está vacío se extraen los dos bytes direccionados por *tail* y *tail+1*, y se incrementa el contador. Además, como después de leer el buffer ya no puede estar lleno, se actualiza la bandera *notfull* a TRUE. Esto se muestra en el Listado 6-9, que corresponde a la parte del programa en que se realiza el proceso de actualización de parámetros (el bucle del diagrama de flujo de la Figura 6.2), pues es aquí el único punto del programa en que se extraen datos del buffer.

```
/*
        set_sidregister(address, value) es la función que actualiza los parámetros del emulador
Variables:
        temp            valor del contador de milisegundos en el ciclo anterior
        mstimer         valor actual del contador de milisegundos
*/

        while(temp==mstimer)                    //mientras no haya pasado 1 ms
        {
                if (!BUFFER_IS_EMPTY)           //si el buffer no está vacío
                {
                        set_sidregister (SPI_buffer_data[SPI_buffer_tail], SPI_buffer_data[SPI_buffer_tail+1]);
                        SPI_buffer_tail= (SPI_buffer_tail+2)%(BUFFER_SIZE);             //incremento
circular
                        SPI_buffer_notfull = 1;
                }
        }
        temp=mstimer;

}//Fin del bucle principal, que contiene los procesos de actualización de parámetros y cálculo de la envolvente.
```

Listado 6-9. Extracción de datos del buffer de recepción del emulador

6.6 Actualización de los parámetros

La función encargada de esto es *set_sidregister*, que recibe como parámetros la dirección y el valor del registro escrito. Lo primero que hace esta función es actualizar el valor en la estructura de registros del SID. Sin embargo, al escribir algunos registros es necesario además actualizar algunas variables que usa el emulador:

- Si se cambia la frecuencia de una voz, hay que calcular además el coeficiente de frecuencia que usará el acumulador de fase para el cálculo de la muestra de audio.
- Si se cambia la forma de onda seleccionada en una voz, o el ancho de pulso de la onda rectangular, hay que volver a calcular el array con la forma de onda de dicha voz. Esto lo hace la función *init_waveform*.
- Si se cambian los parámetros de la curva ADSR de una voz, hay que calcular el nivel de sostenimiento y los incrementos de ataque, caída, y relajación usados para el cálculo de la envolvente. Esto lo hace la función *set_envelope*.

```
uint8_t set_sidregister(uint8_t regnum, uint8_t value)
{
        if(regnum>NUMREGISTERS-1) return 1;
        Sid.sidregister[regnum]=value;          //actualiza el valor en la estructura de registros

        switch(regnum)
        {
                //voice1
                case 1:     //Freq1_HI
                {
                        freq_coefficient[0]=(Sid.sidregister[0]+(Sid.sidregister[1]<<8))/8; break;
                }
                case 3: init_waveform(&Sid.block.voice1);break;     // PW1_HI
                case 4: init_waveform(&Sid.block.voice1);break;     //Control Reg1.
                case 5: setenvelope(&Sid.block.voice1);break;       //AD1
                case 6: setenvelope(&Sid.block.voice1);break;       //SR1

                /*****************************************
                Igual para las voces 2 y 3…
                *****************************************/

        }
return 0;
}
```

Listado 6-10. Fragmento de la función de actualización de parámetros del emulador

La función *init_waveform* recibe como parámetro un puntero a la estructura de una voz (teniendo así acceso fácilmente a los valores de sus registros), y calcula los valores del array de la forma de onda de dicha voz. Sólo lo calcula si la forma de onda se ha modificado, evitando, por ejemplo, volver a calcular toda la forma de onda de una voz siempre que se active o desactive el bit GATE del registro de control, cosa que será habitual.

El array contiene un periodo de la forma de onda de la voz, con 256 muestras, cada una siendo un entero de 8 bits con signo. Las ecuaciones que calculan la forma de onda son: para la onda en diente de sierra la (6.5); para la onda triangular la (6.6); para la onda de pulso la (6.7).

$$wavek[n] = n - 128 \quad para \ \text{n} = 0 \ldots 255 \tag{6.5}$$

$$wavek[n] = \begin{cases} 2 \cdot n - 128 & para\ n = 0 \ldots 127 \\ 2 \cdot (n\tilde{}) - 128 & para\ n = 128 \ldots 255 \end{cases} \tag{6.6}$$

$$wavek\ [n] = \begin{cases} 127\ para\ n < PW \\ -128\ para\ n > PW \end{cases} \quad (6.7)$$

Siendo wavek[256] el array de forma de onda de la voz (esto es, wave0, wave1, ó wave2) y PW el valor de la anchura de pulso expresada en 8 bits. En cuanto a la generación de la onda triangular, se ha llamado n˜ a la inversión bit a bit del número n (cambiar los ceros por unos y los unos por ceros), lo cual es equivalente a calcular (255-n). En el Listado 6-11 se muestra la función *init_waveform* completa.

```
uint8_t init_waveform(struct Voice *voice)
{
        uint16_t n;
        uint8_t wavetype=voice->ControlReg;
        int8_t *wave_array;                     //puntero al array de forma de onda

        n=get_wavenum(voice); //get_wavenum devuelve 0 para la voz 1, 1 para la voz 2, 2 para la voz 3

        if ( (WAVEFORM_IS_EQUAL) && (!WAVEFORM_GOT_RECTANGLE || PW_IS_EQUAL) )
        {       //si se ha escrito el registro de control o PW, pero no hay que cambiar la forma de onda
                return 1;
        }
        CREGS [n]=wavetype;
        PWS[n]=voice->PW;

        if(n==0) wave_array=wave0;
        if(n==1) wave_array=wave1;
        if(n==2) wave_array=wave2;

        for(n=0;n<256;n++)
        {
                *wave_array=0xFF;

                if(wavetype&SAWTOOTH)  {*wave_array&=n-128;}

                if(wavetype&TRIANGLE)
                {
                                if(n&0x80) {*wave_array&=((n^0xFF)<<1)-128;}//n XOR con 255 es n negado
                                else {*wave_array&=(n<<1)-128;}
                }

                if(wavetype&RECTANGLE)
                {
                        if(n>(voice->PW >> 4)) {*wave_array&=127;}
                        else {*wave_array&=-128;}
                }

                wave_array++;
        }
        return 0;
}
```

Listado 6-11. Función *init_waveform*

La función *set_envelope* calcula los parámetros de la envolvente, según las ecuaciones (6.8) a (6.11), que serán utilizados para el cálculo de la envolvente. Primeramente se calculan con precisión de 16 bits sin signo, y luego se convierten a 8 bits. El nivel máximo considerado para el cálculo (16 bits) es 19000, que

al convertir a 8 bits se convierte en 74. Así se ajusta el valor máximo de la mezcla, como se explicó en el capítulo de cálculo de la muestra de audio.

$$m_{attack} = \frac{Nivel_{máx}}{T_{attack}\ (ms)} \tag{6.8}$$

$$m_{decay} = \frac{Nivel_{máx} - Nivel_{sustain}}{T_{decay}\ (ms)} \tag{6.9}$$

$$m_{release} = \frac{Nivel_{sustain}}{T_{release}\ (ms)} \tag{6.10}$$

$$Nivel_{sustain} = \frac{Nivel_{máx}}{15} \cdot Sustain\ (registro) \tag{6.11}$$

Los valores de los tiempos de ataque y de caída/relajación expresados en milisegundos para cada valor de registro, necesarios para el cálculo, se almacenan en dos arrays de 16 elementos cada uno, y el proceso de cálculo se muestra en el Listado 6-12. Por motivo de la falta de espacio en la memoria RAM (en la que se almacenan las variables), y como no se accede constantemente a estos arrays, se ha utilizado una sintaxis especial para guardarlas en la memoria de programa (*flash*): Para almacenar una variable en memoria de programa se utiliza el modificador *PROGMEM*. Para leer esa variable, la función *pgm_read_word* recibe como parámetro un puntero a memoria de programa y devuelve el valor de la palabra almacenada en esa dirección (16 bits).

```
/*Variables:
        m_attack[3]  array con el incremento de la envolvente de cada voz para la fase de ataque
        m_decay[3]  array con el decremento de la envolvente de cada voz para la fase de caída
        m_release[3]        array con el decremento de la envolvente de cada voz para la fase de relajación
        sustain_level[3]    array con el nivel de la envolvente de cada voz para la fase de sustain.
Declaración de los arrays con los tiempos:
        uint16_t AttackRate[16] PROGMEM={2,4,16,24,38,58,68,80,100,250,500,800,1000,3000,5000,8000};
        uint16_t DecayReleaseRate[16] PROGMEM={6,24,48,72,114,168,204,240,300,750,1500,2400,3000,9000,15000,24000};
*/


#define MAXLEVEL  19000
#define SUSTAINFAKTOR ( MAXLEVEL / 15 )

void setenvelope(struct Voice *voice)
{
   uint8_t n;

   n=get_wavenum(voice);                    //0 para la voz 1; 1 para la voz 2; 2 para la voz 3
    attackdecay_flag[n]=TRUE;
    level_sustain[n]=(voice->SustainRelease>>4)*SUSTAINFAKTOR;

   m_attack[n]=MAXLEVEL/pgm_read_word(&AttackRate[voice->AttackDecay>>4]);
   m_decay[n]=(MAXLEVEL-level_sustain[n]*SUSTAINFAKTOR)/pgm_read_word(&DecayReleaseRate[voice-
>AttackDecay&0x0F]);

   m_release[n]=(level_sustain[n])/pgm_read_word(&DecayReleaseRate[voice->SustainRelease&0x0F]);
}
```

Listado 6-12. Cálculo de los incrementos de la curva ADSR

6.7 Inicialización

Este proceso lo realiza la función *Init ()*, que se ejecuta al principio de la función main, y se muestra en el Listado 6-13, y en él se configura el Atmega 168, concretamente:

- Se configura la interfaz SPI, como esclavo, LSB primero. La velocidad de transmisión se configura en el maestro, es decir, la placa de control. Al ser una transmisión síncrona (con señal explícita de reloj) el esclavo se adapta a esa velocidad.
- Configurar el *Timer/Counter 2* para dividir el reloj de sistema (16 MHz) por dos (8 MHz). Esta será la fuente de reloj del *Timer/Counter 1*.
- Configurar el *Timer/Counter 1* para funcionar en modo *Fast PWM, 8 bits*, y para que la fuente de reloj sea externa (generada por el *T*imer*/Counter 2*).
- Activar las interrupciones por recepción de SPI y *overflow* del *Timer/Counter 1*.

Como ya se ha indicado, configurar el Atmega consiste en escribir determinados valores en sus registros de configuración. En las hojas de características del microcontrolador (Atmel Corporation, 2009) se proporciona información detallada sobre qué valores hay que escribir en cada registro para cada configuración.

En las primeras líneas se configuran los pines del chip, como entradas o como salidas. Esta información está contenida en los registros *DDRB, DDRC y DDRD (Data Direction Registers)*, en los que cada bit indica la dirección de cada pin de los puertos B, C y D. Por defecto todos los valores están a 0, significando que los pines funcionan como entrada, y para activarlos como salida hay que escribir su bit correspondiente a 1. El emulador sólo utiliza como salidas la salida de audio PWM (pin 15, PB1) y la salida del *Timer/Counter 2* (pin 5, PD3). Adicionalmente el pin MISO, aunque en el funcionamiento normal no se utiliza, puede ser interesante usarlo para monitorizar el estado del emulador desde la placa de control, y por eso está configurado también como salida. La configuración del módulo contador 1 está en los registros *TCCR1A*, *TCCR1B* y *TCCR1C* (*Timer/counter control registers 1*), y la del módulo contador 2 en *TCCR2A* y *TCCR2B* (*Timer/counter control registers 2*). Además hay que configurar el valor cero para el comparador del contador 2. La configuración del módulo SPI está en el registro SPCR (SPI control register). En último lugar la instrucción *sei ()* habilita las interrupciones.

```
void Init (void)
{

        DDRB |= (1<<PB4);           //MISO->salida
        DDRB |= (1 << PB1);         //OC1A (PWM)-> salida
        DDRD |= ( 1 << PD3);                    // TIMER2: set OC2B to output.

        TCCR1A  =  (1 << WGM10) | (1 << COM1A1) ;       //
        TCCR1B  =  (1 << WGM12) | (1 << CS10);          // FAST PWM (8-Bit) en OC1A, reloj sin escalar
        TCCR1B |= (1 << CS10) | (1 << CS11) | (1 << CS12); //fuente de reloj externa, en el pin T1 (pin 11)

        // TIMER2 usado como divisor de reloj, FCPU/2 (8 MHz)
        TCCR2A = (1 << WGM21);      // TIMER2: modo CTC
        TCCR2B |= (1 << CS20);      //Fuente de reloj -> 16MHz
        TCCR2A |= (1 << COM2B0);    // TIMER2: toogle OC2B on compare match
        OCR2B=0;                                // TIMER2: valor para el comparador

        TIMSK1 |= (1 << TOIE1); //Avtivar interrupción por overflow en timer1

        // SPI: Habilitar SPI, LSB primero. Activa interrupción por recepción de un byte completo. Configurar como esclavo.
        SPCR = ( (1<<SPE) | (1<<SPIE) | (1<<DORD) );
        SPCR &= ~(1<<MSTR);

        sei();      //Activar interrupciones.
}
```

Listado 6-13. Inicialización del emulador

6.8 Programación del Atmega 168

Cuando se habla de programar un microcontrolador, significa introducir en su memoria (flash) el programa que se pretende que ejecute. El código fuente del programa del emulador del SID está incluido en el fichero llamado "*SIDuino_SPI.ino*". La extensión "*.ino*" es la de los ficheros programados en lenguaje de programación Arduino, y es "*.pde"* para versiones antiguas de la IDE de Arduino, si se usara una de éstas sería necesario cambiar la extensión del archivo.

Para programar un Atmega 168 es necesario un dispositivo llamado "programador", que por un lado esté conectado al ordenador, para recibir el programa, y por otro lado al Atmega, para escribirlo en su memoria. Afortunadamente, la misma placa Arduino que se usa para el control del sintetizador puede programarse para ser un programador, no siendo necesario comprar un nuevo dispositivo. Para ello hay que "subir" a la placa el programa llamado "*ArduinoISP*"[1], mediante el siguiente procedimiento:

- Conectar la placa Arduino al ordenador mediante un cable USB
- Abrir la IDE de Arduino
- En la barra de menús, *File->Examples->ArduinoISP*. El programa viene incluido por defecto en la IDE de Arduino.
- En *Tools->Board* seleccionar el modelo concreto de placa Arduino que se utilice.
- En *Tools->Serial Port* seleccionar el puerto del ordenador al que está conectado Arduino.
- Por último, *File->Upload* o el botón con forma de flecha hacia la derecha. Esta instrucción hace que el código se compile y se transmita a la placa Arduino.

La conexión del programador con el Atmega es vía SPI, igual que en el emulador, así que se puede programar directamente desde la placa del shield, siguiendo el siguiente procedimiento:

- Cambiar jumper del shield de la posición "normal" a la posición "programación". Esto conecta el pin RESET del Atmega con el pin 10 de Arduino, lo cual es necesario para el proceso de programación. La conexión SPI ya está hecha en la placa.
- Montar la placa del emulador sobre la placa Arduino.
- Abrir el fichero *SIDuino_SPI.ino* con la IDE de Arduino.
- En *Tools->Programmer* seleccionar *Arduino as ISP*. Esto le indica a la IDE que el programador utilizado es una placa Arduino.
- En *Tools->Board* seleccionar un modelo cualquiera de placa que incluya el Atmega 168, por ejemplo *Arduino Diecimilia or Duemilanove w/ Atmega168.*
- Por último, *File->Upload using programmer*. El programa del emulador ya está en el Atmega 168.

Para que el sistema funcione, hay que volver a cambiar el jumper a la posición "normal". Es recomendable que todos los procesos de conexión o desconexión se hagan cuando la alimentación (USB) no esté conectada. No está de más recordar que en este punto la placa Arduino es ahora un programador, para que vuelva a ser la placa de control del sintetizador hay que subir el programa *SIDuinaster.ino*, por el mismo procedimiento que se subió *ArduinoISP*.

[1] ISP son las siglas de *in-system programming* o *in-system programmer*, y hace referencia a la posibilidad de programar el microcontrolador sin necesidad de sacarlo del circuito en que esté funcionando.

7. RECEPCIÓN DE MIDI

Esta placa se encarga de recibir mensajes MIDI de un controlador como puede ser un teclado de tipo piano, y enviárselos directamente a la placa de control. Se trata de un shield que se monta encima de la placa de generación de sonido, que a su vez está montada sobre la placa de control. El código necesario incluye:

- El fichero *ArdoMIDI.ino* contiene el programa que ejecuta el microcontrolador de esta placa. Está incluido en el CD que acompaña al proyecto.
- La librería *Wire* de Arduino que gestiona la interfaz I^2C del Atmega[1]. Está entre las librerías estándar de Arduino, y se descarga automáticamente con la IDE de Arduino.
- La librería *MIDI* de Arduino, que gestiona la recepción/transmisión de MIDI, a través de la USART del Atmega. A diferencia de *Wire.h*, no está incluída en la IDE, pero se puede descargar del sitio web de Arduino, además de estar incluida en el CD del proyecto.

Como ya se ha dicho, está construido en una placa llamada ArdoMIDI, facilitada por el tutor del proyecto, Lino García, cuyo esquema de circuito se muestra en la Figura 7.1. Es necesario mencionar el conjunto de jumpers S1, que permite conectar de distintas formas la interfaz MIDI, la UART del microcontrolador y los pines laterales. El modo normal de funcionamiento (jumpers en la posición C) conecta la UART del microcontrolador con la interfaz MIDI. Además se utiliza también la posición B que conecta la UART del microcontrolador con los pines laterales Rx y Tx, para programar la placa, como se explica más adelante.

[1] En las hojas de características de los Atmega a esta interfaz se le llama *Two Wire Inteface* (*TWI)*, pero en este documento se hace referencia a ella como interfaz I^2C, porque es perfectamente compatible con dicho protocolo, más extendido.

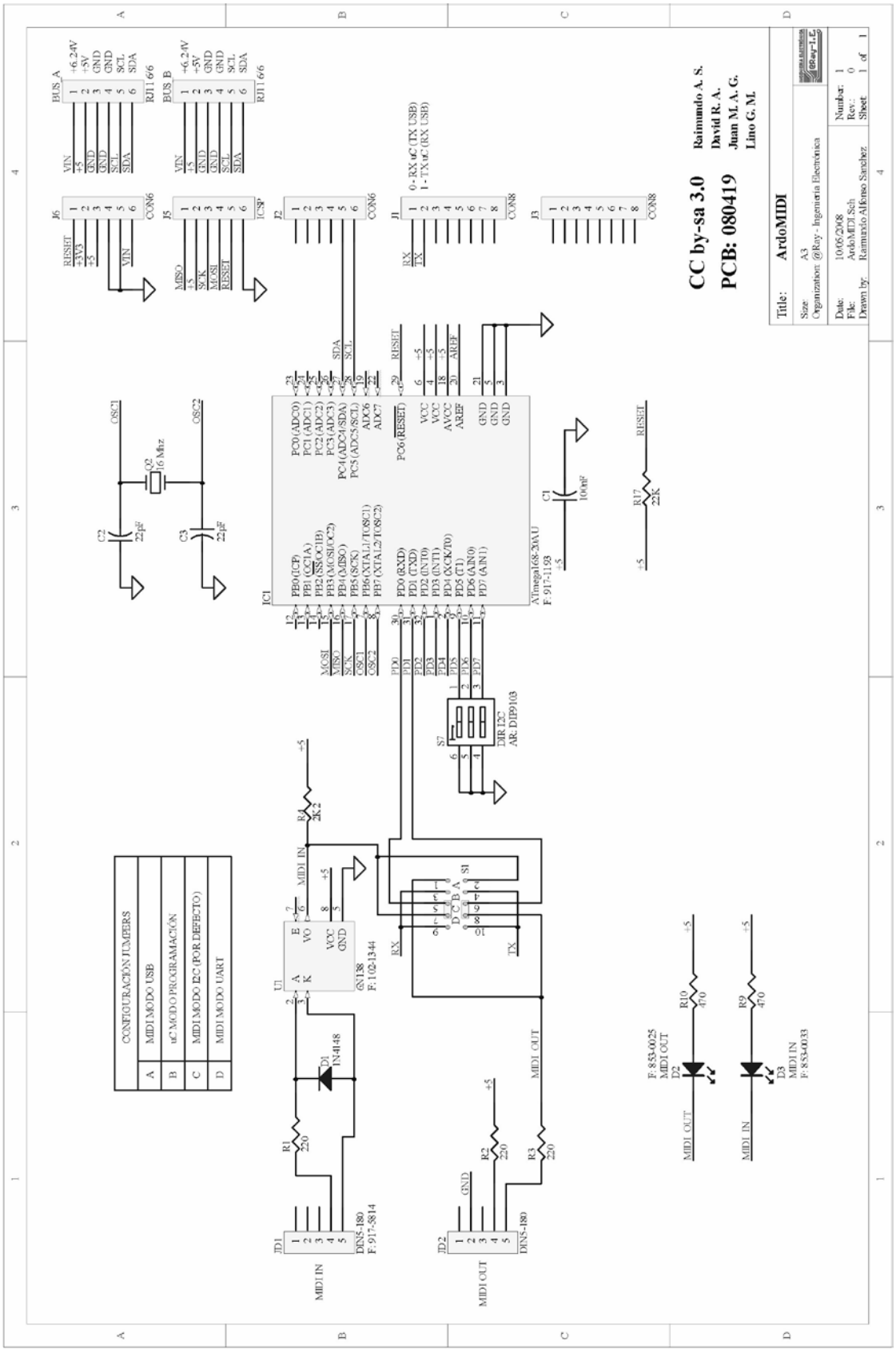

Figura 7.1. Esquema del circuito de la placa ArdoMIDI

7.1 Interconexión

La interfaz usada para interconectar esta placa con la placa de control es I^2C, que utiliza únicamente dos líneas: *SDA* (datos), y *SCL* (reloj). La interconexión genérica entre varios dispositivos por I^2C se muestra en la Figura 7.2. Típicamente al bus I^2C pueden estar conectados un dispositivo maestro, que comienza las transferencias de datos, y varios esclavos. En este caso sólo hay dos dispositivos: ArdoMIDI (Maestro y transmisor), y la placa de control (esclavo y receptor).

Al montar el shield sobre la placa Arduino se establece la conexión I^2C entre ambos. La única electrónica externa necesaria para la conexión son las resistencias de pull-up R1 y R2, que como no están incluidas ni en el shield ardoMIDI ni en la placa Arduino se han colocado en la placa de generación de sonido.

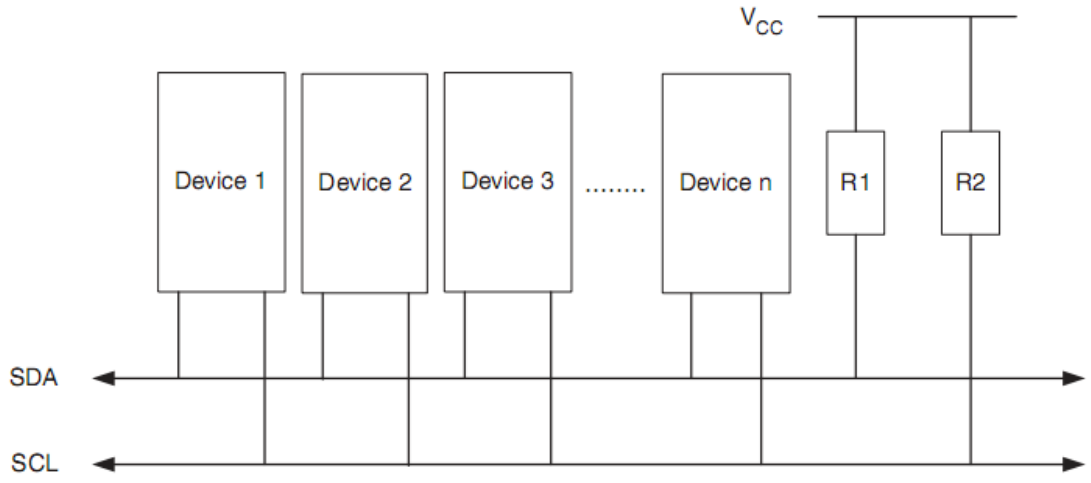

Figura 7.2. Interconexión genérica I^2C

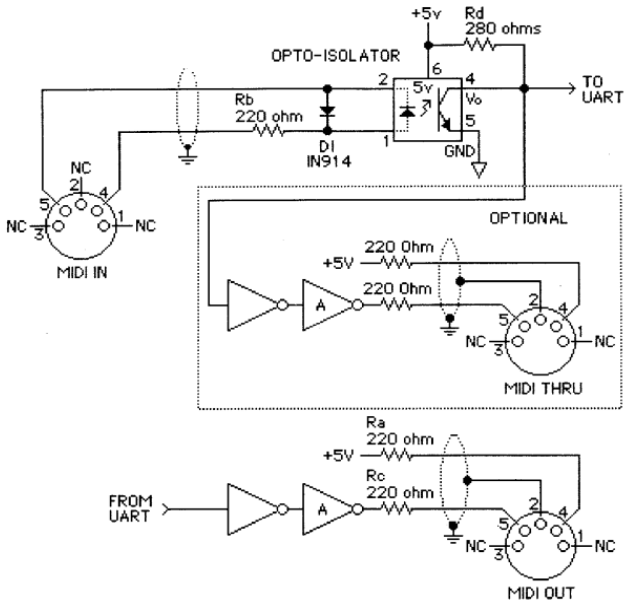

Figura 7.3. Conexión estándar MIDI

En la Figura 7.3 se muestra las conexiones estándar de MIDI, descritas en la espefificación MIDI (MIDI Manufacturers Association, 1995). El microcontrolador Atmega 168 incluye una *USART* (universal synchronous/asynchronous receiver/transmitter), por lo que es perfectamente compatible con MIDI, añadiendo el opto-acoplador y las resistencias externas indicadas en la figura. La placa ardoMIDI incluye las conexiones MIDI IN y MIDI OUT.

7.2 Funcionamiento del programa

En el Listado 7-1 se muestra el comienzo del programa *ArdoMIDI.ino*. En un programa de Arduino siempre hay dos funciones principales: *setup ()*, que se ejecuta una sola vez al iniciar el programa, y *loop ()*, que se ejecuta repetidamente después de *setup ()*. La declaración de funciones puede ir antes o después, pero es típico incluirlas después. El protocolo I^2C es algo más complejo que SPI o USART, pero esto no es problema porque se utiliza la librería *Wire.h* de Arduino, que se encarga de gestionar la interfaz. Para más información sobre este protocolo, en la hoja de características del Atmega 168 (Atmel Corporation, 2009) está detalladamente explicado. En lo que a I^2C se refiere, *Wire.begin ()*[1] es el método que configura la interfaz como maestro. *Wire.begin(direccion_del_esclavo)* sería la sintaxis para configurar la placa como esclavo, como se hace para la placa de control.

```
#include <Wire.h>
#include <MIDI.h>
/*Códigos de los bytes de estado de los mensajes MIDI de canal*/
#define STATUS_BYTE_NOTE_OFF            0x80
#define STATUS_BYTE_NOTE_ON             0x90
#define STATUS_BYTE_POLY_KEY_PRESSURE   0xa0
#define STATUS_BYTE_CONTROL_CHANGE      0xb0
#define STATUS_BYTE_PROGRAM_CHANGE      0xc0
#define STATUS_BYTE_CHANNEL_PRESSURE    0xd0
#define STATUS_BYTE_PITCH_BEND          0xe0

#define I2C_SLAVE_ADDRESS  1                    //dirección I2C de la placa de control

void setup(){
        Wire.begin();    //iniciar I2C como maestro
        MIDI.begin();
        MIDI.setHandleNoteOn (doHandleNoteOn);  //asignación de handlers para recepción de mensajes de canal
        MIDI.setHandleNoteOff(doHandleNoteOff);
        MIDI.setHandleAfterTouchPoly(doHandleAfterTouchPoly);
        MIDI.setHandleControlChange(doHandleControlChange);
        MIDI.setHandleProgramChange(doHandleProgramChange);
        MIDI.setHandleAfterTouchChannel(doHandleAfterTouchChannel);
        MIDI.setHandlePitchBend  (doHandlePitchBend);
}

void loop(){
        MIDI.read();
}
```

Listado 7-1. Inicialización y bucle principal del programa para la recepción de MIDI

[1] Esta es una forma típica de funcionamiento de una librería Arduino. Al incluir *Wire.h* se crea un objeto de C++ llamado *Wire*, que tiene una serie de variables internas y una serie de métodos (funciones propias del objeto). Normalmente no se permite acceder a las variables del objeto directamente, sino sólo a través de sus métodos.

Igualmente el método *MIDI.begin()* inicializa la recepción MIDI. Una vez inicializada, todos los bytes que se reciban a través de la USART se almacenan en un buffer de recepción. Las siguientes instrucciones son para asignar los *handlers* (manejadores) o *callbacks*, que es la forma de funcionar que tiene *MIDI.h*. Esto se puede entender muy bien con el primer ejemplo. Al método *MIDI.setHandleNoteOn* se le pasa como parámetro un puntero a función (*doHandleNoteOn*), que es a la que se llama *handler*, y que está declarada más abajo en el código del programa. Cuando un mensaje recibido se identifique como del tipo *note on* (nota activada) se ejecutará dicha función.

El uso de handlers permite en este caso utilizar las funciones internas de la librería para identificar el mensaje, o detectar si es válido, y definir fácilmente qué hacer cuando se reciba cada tipo de mensaje, simplemente declarando una función en el programa. Si algún tipo de mensaje no tiene handler asociado, sencillamente no se hace nada con él.

El método MIDI.read(), si se han recibido nuevos bytes, inicia el proceso recién descrito: se extraen bytes del buffer de recepción, se identifica si componen un mensaje válido y, si es así, qué tipo de mensaje es, y en función del tipo ejecuta el handler asignado. Esta función debe llamarse cada poco tiempo, para evitar que se pierdan mensajes.

```
void doHandleNoteOn (byte channel, byte note, byte velocity)
{
  channel |= STATUS_BYTE_NOTE_ON;
  uint8_t message_array []= {channel,note,velocity};

  Wire.beginTransmission(I2C_SLAVE_ADDRESS);  //comenzar la transmisión con el esclavo, direccion=1
  Wire.write (message_array,sizeof(message_array));   //prepara los bytes para enviar, metiéndolos en un buffer
  i2c_error = Wire.endTransmission();                 //envía los bytes y termina la transmision
}
```

Listado 7-2. Handler para la recepción de un mensaje de tipo *note on*

En el Listado 7-2 se muestra a modo de ejemplo la función declarada para la recepción de mensajes del tipo *note on*. En un byte de estado MIDI los cuatro bytes altos corresponden al código del tipo de mensaje y los cuatro bytes bajos al canal utilizado. El parámetro *channel* incluye el número de canal, pero no el código de mensaje, por eso se reconstruye el byte de estado completo al principio de la función, para permitir que la placa de control lo use para determinar qué tipo de mensaje ha recibido.

Wire.write recibe como parámetros un puntero a dato de tipo byte, o lo que es lo mismo, entero de 8 bits sin signo (en este caso un array con el mensaje), y el número de bytes a enviar. *Wire.endTransmision()* transmite los bytes almacenados, y devuelve un código de error si se ha producido alguno y 0 si la transferencia tuvo éxito. Se transmite primero el byte de estado, seguido de uno o dos bytes de datos, según el tipo de mensaje.

El resto de handlers son iguales, lo único que hacen es reenviar el mensaje por I^2C a la placa de control. Se puede ver en el Listado 7-1 que hay handlers para todos los mensajes de canal, aunque el sintetizador

sólo reconoce *note on*, *note off*, y *control change*. Los mensajes no reconocidos son ignorados por la placa de control.

7.3 Programación de la placa

En primer lugar hay que descargar la librería *MIDI.h*, disponible en el sitio web de Arduino (Arduino Playground). La librería se descarga como fichero comprimido en formato *zip*, que simplemente hay que descomprimir en la carpeta de librerías de Arduino, esto es: *Arduino-00xx/libraries*.

El proceso para programar la placa ArdoMIDI es diferente si el microcontrolador tiene cargado en memoria el *boot-loader* de Arduino o no. Este *boot-loader* es un pequeño programa que se almacena en las primeras posiciones de la memoria del microcontrolador (la sección *boot*), y que permite programarlo a través de la *USART*, en lugar de vía SPI. Esto es precisamente lo que utiliza Arduino para poder subir código al microcontrolador a través de la conexión USB. Si el microcontrolador ya tiene el *boot-loader* cargado el proceso puede ser el que sigue:

- Cambiar ambos jumpers del selector (ver Figura 7.1) a la posición B ("uC modo programación"). Esto conecta la USART del microcontrolador a los pines 1 y 2 de la placa Arduino.
- Extraer manualmente el microcontrolador de la placa Arduino y montar el shield ArdoMIDI sore ella. Así el microcontrolador de ArdoMIDI queda conectado a la placa Arduino, y la USART a la interfaz USB (de cara al ordenador es como si el microcontrolador de ardoMIDI fuera el microcontrolador de la placa Arduino).
- Conectar la placa Arduino al ordenador por el cable USB.
- Abrir *ArdoMIDI.ino* con la IDE de Arduino.
- En *Tools->Board* seleccionar el modelo concreto de placa Arduino que se utilice, con el microcontrolador usado por ArdoMIDI.
- En *Tools->Serial Port* seleccionar el puerto del ordenador al que está conectado Arduino.
- *File->Upload* o el botón con forma de flecha hacia la derecha.
- Desmontar la placa y volver a insertar el microcontrolador en Arduino.

Si el Atmega no tiene el bootloader es necesario programarlo vía SPI, como se hizo con el apartado 6.8, con la dificultad añadida de que en ArdoMIDI la interfaz SPI del microcontrolador no está conectada a los pines laterales de la placa, sino únicamente al conector *ICSP (in circuit serial programming)*. ICSP es un conector pensado para la programación de microcontroladores, y ofrece conexión a los terminales del microcontrolador necesarios para programarlo. El conector consiste en 6 pines macho dispuestos como se muestra en la Figura 7.4.

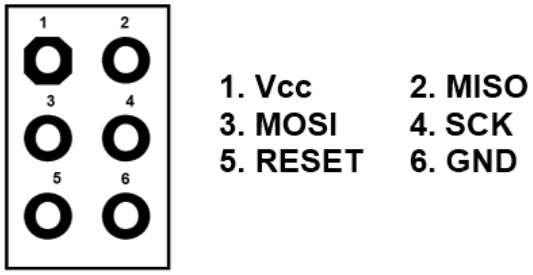

Figura 7.4. Configuración de pines del conector ICSP, vista desde arriba

Para programar el microcontrolador hay que subir el programa *ArduinoISP* a la placa Arduino, y conectarla manualmente con la placa ArdoMIDI vía ICSP, como se muestra en la Tabla 7-1. Estando las placas correctamente conectadas, desde la IDE de Arduino, seleccionando en la barra de herramientas *Tools->Burn Bootloader* se cargará en el microcontrolador el bootloader de Arduino, y a partir de ese momento ya se puede programar la placa a través de USB, siguiendo el procedimiento descrito anteriormente. Si además se abre el fichero *ArdoMIDI.ino* y se selecciona *Tools->Upload using programmer* el código queda introducido en la placa.

Tabla 7-1. Conexión con el microcontrolador Atmega

PIN ARDUINO	PIN ICSP (ArdoMIDI)	PIN Atmega 168 (ArdoMIDI)
D12 (MISO)	1	MISO
D13 (SCK)	3	SCK
D10 (SS)	5	RESET
5V	2	Vcc
D11 (MOSI)	4	MOSI
GND	6	GND

8. CONTROL DEL SINTETIZADOR. PLACA PRINCIPAL

La placa de control consiste en la placa Arduino Duemilanove, utilizando los siguientes ficheros, todos ellos contenidos en el CD que acompaña al proyecto.

- *SIDuinaster.ino* es el programa principal que ejecuta la placa.
- *Message_buffer.h* contiene la implementación del buffer de mensajes, con los procesos para almacenar y extraer mensaje. Debe estar en la misma carpeta que el programa principal.
- La librería *SID* de Arduino, diseñada originalmente para el proyecto *SIDaster*, que gestiona toda la interacción con el SID: generación de reloj y escritura de registros. Debe guardarse en la carpeta de librerías de la IDE de Arduino.

La placa de control se encarga procesar los mensajes MIDI provenientes del controlador MIDI (por ejemplo un teclado) y del puerto USB. Esto se hace en dos etapas principales: recepción/almacenado de mensajes, lectura/procesado de mensajes. Un diagrama de bloques funcional del programa se muestra en la Figura 8.1.

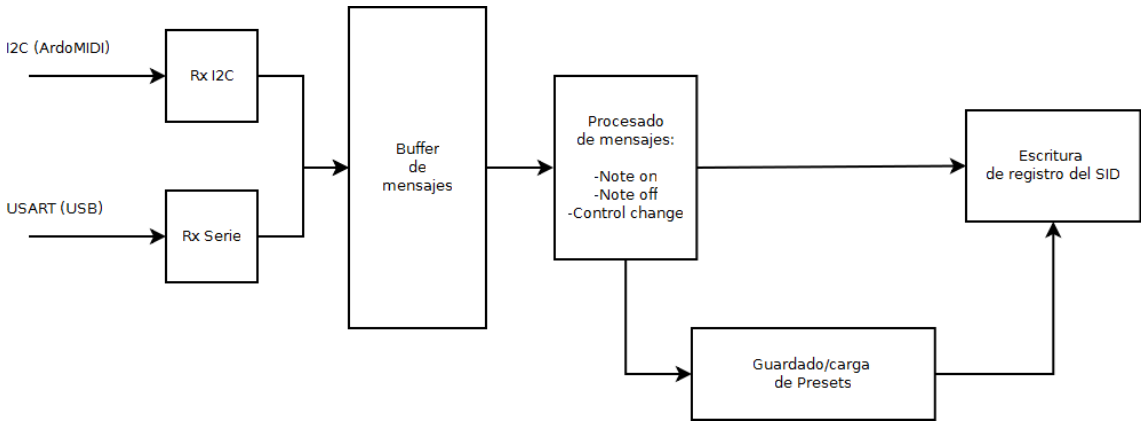

Figura 8.1. Diagrama de bloques functional del programa principal de la placa de control

La idea es parecida a la recepción del emulador: cuando llega un mensaje se almacena en el bufer FIFO, y constantemente se van extrayendo mensajes y procesándolos. Dado que quien genera sonido es el SID (o el emulador), y la forma de controlarlo es escribiendo en sus registros, el objetivo final del procesado de mensajes es decidir qué registros del SID se escriben en función de los mensajes recibidos y del estado del sintetizador, y actualizar dicho estado.

8.1 Comunicación con el SID

En este apartado se describe la librería *SID.h*, creada originalmente para el proyecto *SIDaster* (Banson), ya que el programa de la placa de control está basado en el programa principal de dicho proyecto. Esta librería contiene todo lo necesario para comunicarse con el SID, conectado a la placa Arduino mediante el circuito de SIDaster.

La libería incluye los ficheros *SID.h*, *SID.*c y *6581.h*. El fichero *6581.h* define nombres de registros del SID, bits de los registros, para poder acceder a ellos fácilmente, por ejemplo *SID_CTRL1* para el registro de control de la voz uno o *SID_GATE* para el bit *GATE* de dicho registro (consiste en un byte con todos los bits a 0 menos el referenciado). En *SID.h* y *SID.cpp* se define una clase llamada SID, que incluye tres métodos: configuración del reloj, configuración de SPI, y escribir un registro del SID, como se muestra en el Listado 8-1.

```
class SID
{
        public:
                void clk_setup(void);           //1MHz clock on pin 9 of the Arduino.
                void SPI_setup(void);                   //Pin7-> ~CS para el SID. SPI Maestro, 0,5 Mhz, LSB primero
                void send(byte address, byte data);
};
```

Listado 8-1. Declaración de la clase SID

Con la configuración del reloj se pretende obtener una señal cuadrada de 1 MHz. Para ello se utiliza el *Timer/Counter 1* del Atmega, configurado en modo *CTC* (*Clear timer on compare match)*: En el registro *OCR1A* se almacena un número que será el tope del contador. El contador (registro *TCNT1*) se incrementa en 1 a cada ciclo de reloj, y constantemente se compara su valor con el almacenado en *OCR1A* (recuérdese el diagrama de bloques de un módulo contador, en la Figura 6.4). El modo *CTC* consiste en que cada vez que el valor de ambos registros coincide el contador se reinicia, es decir, se pone a cero. Por lo tanto, la frecuencia de reinicio del contador corresponde a la expresión (8.1).

$$F_{contador} = \frac{F_{clk}}{OCR1A + 1} \tag{8.1}$$

Por otro lado, el generador de forma de onda se configura para que cada vez que el valor de ambos registros coincida se conmute la salida, es decir: si estaba a nivel bajo se pone a nivel alto, y si estaba a nivel alto se pone a nivel bajo. De este modo, la salida es una onda cuadrada de la mitad de la frecuencia del contador. Si se selecciona F_{clk} = 16 MHz, y OCR1A=7 , se tiene una señal cuadrada con frecuencia según la ecuación (8.2). La configuración del *Timer/Counter 1* se hace escribiendo los valores pertinentes en los registros TCCR1A y TCCR1B (*Timer/counter 1 control registers*), como se muestra en el Listado 8-2.

$$F_{out} = \frac{F_{clk}}{2 \cdot (OCR1A + 1)} = 1\ MHz \tag{8.2}$$

```
void SID::clk_setup(void)
{
        pinMode(SID_ARDUINO_MOS_CLK,OUTPUT);                    //pin 9 de Arduino como salida
        TCCR1A &= ~((1<<COM1A1) | (1<<COM1A0) | (1<<WGM11) | (1<<WGM10));
        TCCR1B &= ~((1<<WGM13) | (1<<WGM12) | (1<<CS12) | (1<<CS11) | (1<<CS10)); //reiniciar los registros.
        TCCR1A |= (0<<COM1A1) | (1<<COM1A0);          //modo CTC
        TCCR1A |= (0<<WGM11) | (0<<WGM10);
        TCCR1B |= (0<<WGM13) | (1<<WGM12);                  //configurar el generador de forma de
onda
        TCCR1B |= (0<<CS12) | (0<<CS11) | (1<<CS10);        //frec. de reloj 16 MHz
        OCR1A = 7;
}
```

Listado 8-2. Configuración del *Timer/counter 1* como reloj de 1 MHz para el SID

En cuanto a la configuración de SPI, se configura la interfaz para que funcione como maestro, que la transferencia comience por el LSB, y que la velocidad de transmisión sea de 0,5 Mbits/s. La configuración de la interfaz SPI se hace mediante el valor del registro *SPCR* (*SPI control register*).

```
void SID::SPI_setup(void)
{
        pinMode(SID_ARDUINO_MOS_CS,OUTPUT);
        digitalWrite(SID_ARDUINO_MOS_CS, HIGH);
        pinMode(SID_ARDUINO_SPI_RCK,OUTPUT);
        digitalWrite(SID_ARDUINO_SPI_RCK, HIGH);
        pinMode(SID_ARDUINO_SPI_MOSI,OUTPUT);
        pinMode(SID_ARDUINO_SPI_CLK,OUTPUT);
        SPCR |= ( (1<<SPE) | (1<<MSTR) );           // enable SPI as master
        SPCR |= (1<<DORD);//LSB First

        SPCR |= (1<<SPR1); SPSR |= (1<<SPI2X);      //0,5MHz

        SID_clr=SPSR;
        SID_clr=SPDR;
}
```

Listado 8-3. Configuración de la interfaz con el SID: SPI y Chip Select

El método que escribe un registro en el SID es SID.*send* que se muestra en el Listado 8-4, que recibe como parámetros la dirección y el valor del registro. *SPDR* (*SPI data register*) es el registro de datos de la interfaz SPI, y su valor es el que se envía en la transferencia SPI. Cuando se escribe un valor en dicho registro, automáticamente se comienza la generación de la señal de reloj *SCK*, y por lo tanto la transferencia (el funcionamiento de una transferencia SPI está explicado en la sección 5.2).

Cuando la transferencia termina, el bit *SPIF* (*SPI flag)* del registro SPSR (*SPI status* register) se pone a 1, y se puede escribir un nuevo valor. La variable *SPI_clr* se utiliza únicamente porque esta bandera se pone a 0 automáticamente al leer el registro de datos de SPI.

```
void SID::send(byte address, byte data)
{
	PORTD |= 128;		//desctivar el chip select del SID, en el pin 7 de Arduino
	SPDR = address;		//enviar número de registro y esperar a que termine
	loop_until_bit_is_set(SPSR, SPIF);
	SID_clr=SPDR;
	SPDR = data;		//enviar valor de registro y esperar a que termine
	loop_until_bit_is_set(SPSR, SPIF);
	SID_clr=SPDR;
	PORTB &= ~(4);
	PORTB |= 4;		//pulso en RCK (pin  10 de Arduino). activa la salida paralelo de los registros 74HC595
	PORTD &= ~(128);	//activa el chip select del SID
	delayMicroseconds(300);
}
```

Listado 8-4. Método para la escritura de un registro del SID

8.2 Buffer de mensajes

El buffer de mensajes está implementado en un fichero de cabecera llamado *message_buffer.h*, que contiene además la definición del tipo de estructura *TchannelMessage*. El buffer es exactamente igual que del emulador del SID, descrito en la sección 6.5, con la diferencia de que en vez de almacenar datos de tipo *byte* almacena datos de tipo *TchannelMessage*, es decir, en cada posición del buffer se almacena un mensaje. En este caso el tamaño del buffer es de 16 mensajes. En las secciones de código siguientes se muestra la implementación del buffer.

```
typedef struct TchannelMessage {
	byte stat;
	byte data1;
	byte data2;
};
```

Listado 8-5. Declaración del tipo de estructura para mensajes de canal

```
#define MESSAGE_BUFFER_SIZE		16

TchannelMessage message_buffer [MESSAGE_BUFFER_SIZE]={0};
int8_t message_buffer_head=0;
int8_t message_buffer_tail=0;
int8_t message_buffer_notfull=1;//inicialmente vacío
```

Listado 8-6. Declaración del buffer de mensajes

```
inline void buffer_store (TchannelMessage msg)
{
	if (message_buffer_notfull != 0)
	{
		message_buffer [message_buffer_head]=msg;
		message_buffer_head = (message_buffer_head+1) % MESSAGE_BUFFER_SIZE;
		message_buffer_notfull = message_buffer_head - message_buffer_tail;
	}
}

TchannelMessage buffer_read ()
{
	TchannelMessage msg;
	msg.stat=0;
	if ( (message_buffer_head != message_buffer_tail) || (message_buffer_notfull==0))	//si no esta vacio
	{
		msg=message_buffer[message_buffer_tail];
		message_buffer_tail = (message_buffer_tail+1) % MESSAGE_BUFFER_SIZE;
	}
	return msg;
}
```

Listado 8-7. Funciones de acceso al buffer de mensajes: almacenar y extraer mensaje

8.3 Recepción I²C

Todos los mensajes MIDI de canal recibidos de la interfaz MIDI llegan a la placa de control pasando por la placa de recepción MIDI, por el protocolo I^2C. Al igual que en la placa ArdoMIDI, la interfaz I^2C se gestiona usando la librería *Wire.h* de Arduino. En el Listado 8-8 se muestra la inicialización de la interfaz. Para configurar la interfaz como dispositivo esclavo el método *Wire.begin* debe recibir como parámetro la dirección I^2C, que es en este caso 1.

```
void i2c_init(byte address)
{
	Wire.begin(address);
	Wire.onReceive(i2c_receive);
}
```

Listado 8-8. Inicialización de la interfaz I^2C en la placa de control.

La siguiente línea de código funciona de igual manera que los *handlers* de la librería *MIDI.h*: la función *i2c_receive*, declarada más abajo, será la que se ejecute cuando se complete una transferencia. Esta función se muestra en el Listado 8-9, y lo único que hace es extraer el mensaje y almacenarlo en el buffer de mensajes. El parámetro *numberOfBytes* es el número de bytes recibidos. Cuando se van recibiendo bytes, la librería *Wire.h* va guardándolos automáticamente en un buffer FIFO. El método *Wire.read()* extrae un byte de dicho buffer. La estructura TchannelMessage tiene un byte de estado y dos bytes de datos. Cuando el mensaje recibido sólo tiene un byte de datos (por ejemplo *Program Change*) el segundo se almacena con valor cero.

```
void i2c_receive (int numberOfBytes)
{//extrae un mensaje y lo almacena en el buffer
        static TchannelMessage channelMessage;
        if (numberOfBytes==3)
        {
                channelMessage.stat=Wire.read();
                channelMessage.data1=Wire.read();
                channelMessage.data2=Wire.read();
        }
        else if (numberOfBytes==2)
        {
                channelMessage.stat=Wire.read();
                channelMessage.data1=Wire.read();
                channelMessage.data2=0;
        }

        buffer_store (channelMessage);
}
```

Listado 8-9. Función que recibe y almacena los datos por I^2C

8.4 Recepción serie

Del ordenador también pueden llegar mensajes MIDI a través de la conexión USB que deben ser almacenados. En este caso no se ha podido implementar usando la librería *MIDI.h*, así que se gestiona la recepción de la *USART* manualmente. Un posible motivo es que este programa utiliza muchas variables, ocupando gran cantidad de memoria RAM, y el buffer utilizado por dicha librería es demasiado grande para caber. Las funciones utilizadas para la recepción de datos por la *USART* son:

- *serial_init (unsigned long baud)*: configuración de la USART exactamente igual que el método *Serial.begin*[1] de Arduino.
- *ISR (USART_RX_vect)*: rutina de interrupción que se ejecuta al recibir un byte completo a través de la USART.
- parseSerialByte (c): procesa los bytes recibidos uno a uno, tratando de reconstruir el mensaje MIDI del que forman parte para almacenarlo en el buffer.

La función *serial_init* se muestra en el Listado 8-10. Básicamente lo que hace es configurar la *USART* a la velocidad indicada por el parámetro *baud*, y habilitar la recepción y la transmisión. La velocidad de transmisión se controla por el número de 16 bits contenida en los registros *UBRR0L* y *UBRR0H* (*USART0 baud rate registers*, byte bajo y alto), y lo que hace la función es calcular el valor de dichos registros para obtener la velocidad dada por el parámetro *baud*.

Además la información sobre el modo de funcionamiento se encuentra en los registros *UCSR0A*, *UCSR0B* y *UCSR0C* (*USART0 control and status registers*). En concreto el bit U2X0 multiplica la velocidad de

[1] El objeto *Serial* de Arduino gestiona las transmisiones a través de la USART, y viene incluida con la IDE Arduino.

transmisión por dos, y según si está activado o no el cálculo será diferente. La función primero intenta calcular el valor con ese bit activado, y si el valor obtenido no es válido lo vuelve a intentar con el bit desactivado.

En las últimas líneas se habilitan la transmisión, la recepción y la interrupción por recepción de un byte a través de la USART. También se inhabilita la interrupción por byte completo transmitido. Configurado así, cada vez que se reciba un byte completo a través de la USART se ejecuta la ISR que gestiona el byte recibido, tratando de reconstruir y almacenar el mensaje MIDI.

La velocidad de transmisión que se ha utilizado es 57600 baudios. En el ordenador se usará algún software para poder enviar MIDI a través de USB, y dicho software deberá configurarse a esta misma velocidad de transmisión.

```
void serial_init (unsigned long baud)
{//copiada del método Serial.begin, de HardwareSerial.cpp, librería estándar de Arduino
	uint16_t baud_setting;
	boolean use_u2x = true;

	#if F_CPU == 16000000UL
		// hardcoded exception for compatibility with the bootloader shipped
		// with the Duemilanove and previous boards and the firmware on the 8U2
		// on the Uno and Mega 2560.
		if (baud == 57600) {
		use_u2x = false;
		}
	#endif
	try_again:

	if (use_u2x) {
		UCSR0A = 1 << U2X0;
		baud_setting = (F_CPU / 4 / baud - 1) / 2;
	}
	else {
		UCSR0A = 0;
		baud_setting = (F_CPU / 8 / baud - 1) / 2;
	}

	if ((baud_setting > 4095) && use_u2x)
	{
		use_u2x = false;
		goto try_again;
	}

	// assign the baud_setting, a.k.a. ubbr (USART Baud Rate Register)
	UBRR0H = baud_setting >> 8;
	UBRR0L = baud_setting;

	UCSR0B |= (1<<RXEN0) | (1<<TXEN0) | (1<<RXCIE0);
	UCSR0B &= ~(UDRIE0);
}
```

Listado 8-10. Función Serial_init

En el Listado 8-11 se muestra la *ISR* que se ejecuta al recibir un byte por la USART. La función *parseSerialByte (c)* procesa el byte recibido, tratando de reconstruir y almacenar el mensaje recibido.

Devuelve un valor lógico que vale FALSE si todavía no se ha reconstruido el mensaje, y TRUE si se ha almacenado o si no se trata de un mensaje válido. Por lo tanto, sólo si el valor devuelto es FALSE la ISR espera a recibir el siguiente byte para continuar el procesado.

```
SIGNAL(USART_RX_vect)
{ //sólo se sale del bucle si se ha completado el mensaje o ha habido error en la transmisión.
        unsigned char c;
        while (1)
        {
                c  =  UDR0;                    leer bit recibido

                if ( parseSerialByte(c) ) break;
                loop_until_bit_is_set(UCSR0A,RXC0);
        }
}
```

Listado 8-11. ISR para la recepción de un byte por la USART

Cuando se recibe un mensaje el objetivo es que si es de canal se almacene en el buffer para su posterior procesado, y si no lo es, se ignore. Por ello cuando se recibe un byte por el puerto serie se debe procesar ese byte para determinar de qué tipo de mensaje forma parte. En lo que al sintetizador construido concierne, el byte recibido puede ser:

- Un **byte de estado** (1xxx xxxx) de un mensaje de canal con un byte de datos
- Un **byte de estado** (1xxx xxxx) de un mensaje de canal con dos bytes de datos
- Un **byte de estado** (1xxx xxxx) de un mensaje que no es de canal, y por lo tanto se considera no válido
- Un **byte de datos** (0xxx xxxx)

Para determinar el estado del receptor se utilizan dos variables enteras: *counter*, el número de bytes válidos recibidos, y *expected* el número de bytes que se esperan recibir. La variable *expected* puede valer 2 o 3, según el tipo de mensaje de canal. En la Figura 8.2 se muestra el diagrama de flujo de la función *parseSerialByte (c)*.

Los bytes recibidos se van guardando en una estructura *TchannelMessage* llamada *msg*, y cuando está el mensaje completo se almacena en el buffer y se reinicia el contador a 1. Esto se hace así para admitir *running status*: en principio se asume que ya se ha recibido el byte de estado. De ese modo si se reciben bytes de datos a continuación, se almacenan como el mismo tipo de mensaje, y si se recibe de nuevo el byte de estado se vuelve a guardar.

Nótese que cuando se completa y almacena un mensaje, se reinicia el contador a 1, en lugar de a 0, como parece lógico. Esto se hace para admitir la función *running status* (estado en curso), que permite al transmisor, si se envían muchos mensajes del mismo tipo seguidos, mandar el byte de estado una sola vez para conseguir más eficiencia. En este receptor, por defecto se asume que el byte de estado ya se ha recibido y está almacenado en el byte de estado de la variable *msg*. Así, si posteriormente se reciben bytes

de datos se asume que son de mensajes válidos del mismo tipo. Cuando se recibe un byte de estado, se sustituye el antiguo y continúa el proceso.

La estructura del diagrama se implementa mediante una estructura *if... else if*, y el tipo de mensajes se determina comparando los cuatro bits altos del byte de estado con el código de cada tipo de mensaje de canal.

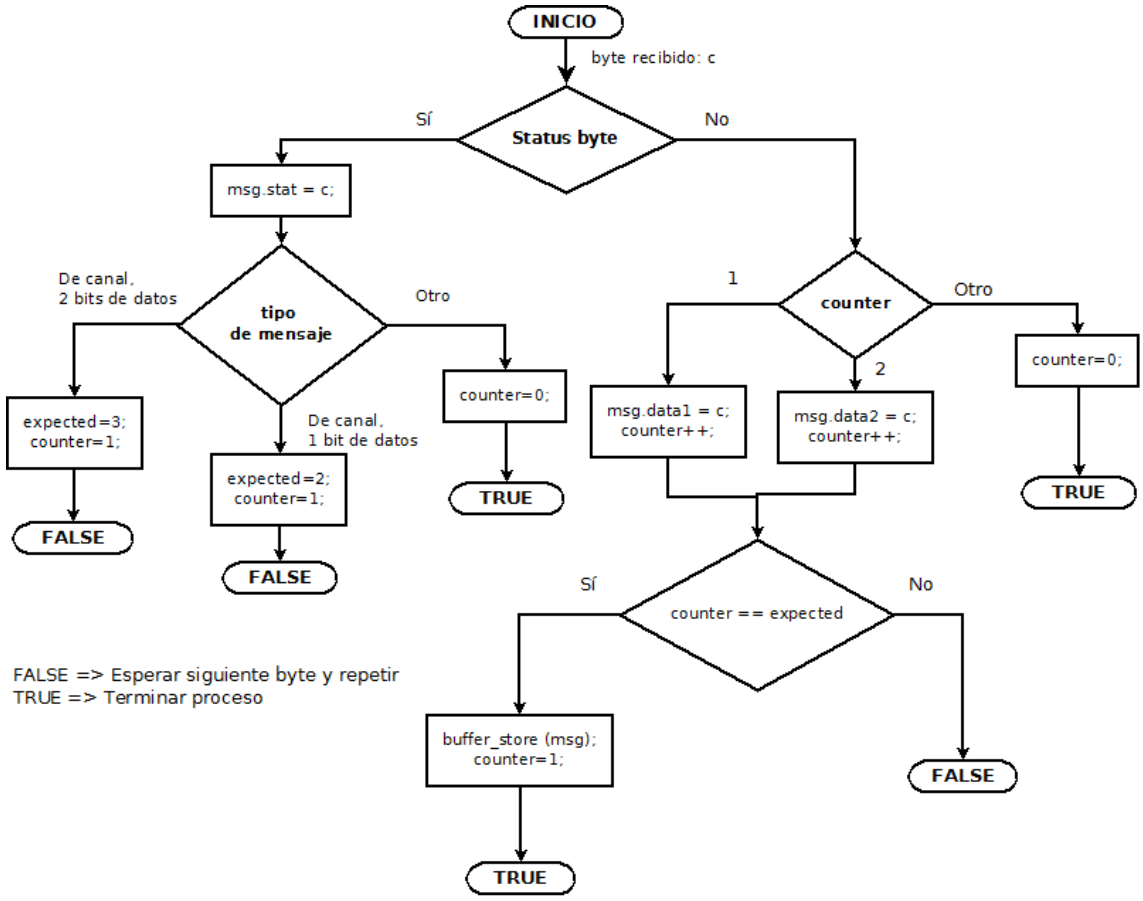

Figura 8.2. Diagrama de flujo de la función *parseSerialByte(c)*

8.5 Programa principal

En el Listado 8-12 se muestra el programa principal de la placa de control. Además de las funciones ya explicadas, es importante la función *synth_init()* que escribe valores de todos los registros (de escritura) del SID, de tal manera que no es necesario configurar el sintetizador para comenzar a utilizarlo. Los parámetros iniciales son: modo polifónico, forma de onda triangular, tiempos de ataque, caída y relajación mínimos, nivel de sustain máximo, volumen máximo y sin filtrar las voces.

```
void setup()
{
        // Configurar la interfaz con el SID
        sid.clk_setup();
        sid.SPI_setup();

        synth_init();

        //configurar las comunicaciones
        i2c_init(I2C_SLAVE_ADDRESS);
        serial_init(SERIAL_MIDI_BAUD_RATE);//57600 baudios
}

// Main program loop
//
void loop()
{
        static TchannelMessage channelMessage;
        if (MESSAGE_BUFFER_AVAILABLE)    //si el buffer no está vacío
        {
                channelMessage = buffer_read();
                proccessMessage (channelMessage);
        }
}
```

Listado 8-12. Programa principal de la placa de control

Los valores de todos los registros del SID se almacenan en el sintetizador se almacenan en la placa de control en una serie de variables globales que se muestran en la Tabla 8-1. *synth_init()* lo único que hace es volcar estas variables en los registros del SID, mediante el método *sid.send (dirección, valor)*. Según en qué modo esté el sintetizador (monofónico/polifónico) se vuelcan las variables relacionadas con un modo o el otro. De esta tabla además se puede ir intuyendo una idea del sintetizador: en modo polifónico todas las voces deben tener los mismos parámetros, es decir, todos los registros salvo los de frecuencia deben tener el mismo valor, y por eso se utiliza una sola variable cuando en monofónico se usan tres.

Tabla 8-1. Variables globales para los registros del SID

Registros del SID		Variables globales		
		Monofónico	Polifónico	Valor inicial
Registros de las voces	FreqHi/Lo	freq1,freq2,freq3		$1CD6_H$ (440 Hz)
	PWHi/ Lo	pw1, pw2, pw3	pw_poly	0800_H
	Control register	ctrl1, ctrl2, ctrl3	ctrl_poly	10_H (TRI)
	Attack/decay	ad1, ad2, ad3	ad_poly	00_H
	Sustain/release	sr1, sr2, sr3	sr_poly	$F0_H$
Registros del filtro	FCHi/Lo	freq	freq_poly	$07FF_H$ (máx)
	RES/Filt	res	res_poly	00_H
	Mode/Vol	mode	mode_poly	$3F_H$

Al igual que en el emulador, constantemente se comprueba si hay mensajes nuevos en el buffer, y se van extrayendo y procesando con la función *proccessMessage (channelMessage)*, que se muestra en el Listado 8-13. Según el tipo de mensaje que sea, se llama a una función que maneja dicho mensaje.

```
void proccessMessage (TchannelMessage message)
{
	static byte status_code, channel;
	status_code = message.stat & 0xF0;
	channel=message.stat & 0x0F;

	switch (status_code)
	{
		case STATUS_BYTE_NOTE_ON:
			DoHandleNoteOn (channel, message.data1, message.data2);
		break;
		case STATUS_BYTE_NOTE_OFF:
			DoHandleNoteOff (channel, message.data1, message.data2);
		break;
		case STATUS_BYTE_CONTROL_CHANGE:
			DoHandleControlChange (channel, message.data1, message.data2);
		break;
	}
}
```

Listado 8-13. Función *processMessage*

8.6 Note on y note off en modo monofónico

El modo de funcionamiento del sintetizador se almacena en una variable global llamada *synth_mode*, que puede valer las constantes MODE_MONO o MODE_POLY (1 y 2, respectivamente). Al comienzo de las funciones de procesado de *note on* y *note off* hay una estructura *if...else if* para procesar el mensaje de modo diferente según el modo.

El comportamiento del sintetizador en modo monofónico es bastante sencillo: cuando llega un mensaje *note on* (con velocidad distinta de cero) se activan todas las voces del SID a la nota recibida. Cuando se recibe un mensaje *note off* o *note on* con velocidad cero, si la nota recibida es la misma que la nota que se está tocando, se silencian las tres voces. Para ello se utiliza la variable *current_note*, que contiene en todo momento el número de nota que se está tocando en el sintetizador. En el Listado 8-14 se muestra un fragmento de la función *DoHandleNoteOn*, que muestra todo este comportamiento, utilizando las funciones *synth_on* y *synth_off*. El comportamiento para mensajes *note off* es exactamente el mismo que para mensajes *note on* con velocidad cero.

La configuración de cada voz es independiente, de tal manera que cada una puede tener parámetros muy diferentes: forma de onda, curva ADSR, filtrada o sin filtrar. Además en modo monofónico se permite usar las modulaciones entre voces, que en modo polifónico no tiene sentido. Otro parámetro independiente para cada voz es el control de la afinación: afinación gruesa (*coarse*) y fina *(fine*). La afinación gruesa actúa como transpositor en semitonos, permitiendo además que cada voz toque una nota diferente a un intervalo fijo (octavas, quintas, etc.). La combinación de transposición, formas de onda distintas y modulaciones entre voces incrementa en mucho las posibilidades tímbricas del sintetizador, a costa de poder interpretar únicamente una nota simultáneamente.

```
void DoHandleNoteOn (byte channel, byte note, byte velocity) {
	if (synth_mode==MODE_MONO)
	{
		if( velocity==0 )
		{
			if (note==current_note)
			{
				current_note=0xFF;
				synth_off (note);
			}
		}
		else
		{
			synth_on(note);
			current_note = note;
		}

		}
	else if (synth_mode==MODE_POLY)
	{
/*************************************
procesado para el modo polifonico…
*************************************/
	}
}
```

Listado 8-14. Procesado de *note on* en modo monofónico

El procedimiento para tocar una nota en una voz es, tanto para el modo monofónico como el polifónico:

- Escribir a 0 el bit GATE de su registro de control (el LSB)
- Calcular el valor de los registros de frecuencia para esa nota, teniendo en cuenta la configuración de afinación de la voz (*coarse* y *fine*).
- Escribir el valor calculado en los registros del SID.
- Escribir a 1 el bit GATE del registro de control.

Para silenciar una voz lo único que hay que hacer es escribir a 0 el bit GATE del registro de control, y a current_note asignarle un valor no válido. El valor FF_H (255) está fuera del rango del SID (y también de MIDI, la nota más aguda es 127) se utiliza este valor para saber que el sintetizador no está tocando ninguna nota. En otro orden de cosas, en el Listado 8-14 se puede observar el detalle de que siempre antes de escribir un registro del SID se actualiza la variable correspondiente, teniendo en todo momento el sintetizador conocimiento del estado del SID.

```
/*Variables:
        ctrl1,ctrl2,ctrl3       los registros de control de cada voz
        freq1,freq2,freq3       los registros de frecuencia (16bits) de cada voz.
        coarse1,coarse2,coarse3         la afinación gruesa para cada voz
        fine1, fine2,fine3              la afinación fina para cada voz
*/
void synth_on (int note)
{
        if (ctrl1 & 1) sid.send(SID_CTRL1,ctrl1 & 0xFE);
        if (ctrl2 & 1) sid.send(SID_CTRL2,ctrl2 & 0xFE);
        if (ctrl3 & 1) sid.send(SID_CTRL3,ctrl3 & 0xFE);

        freq1=SID_freq_number (note, coarse1, fine1);
        sid.send(SID_FREQ1LO,char(freq1));
        sid.send(SID_FREQ1HI,char(freq1>>8));

        freq2=SID_freq_number (note, coarse2, fine2);
        sid.send(SID_FREQ2LO,char(freq2));
        sid.send(SID_FREQ2HI,char(freq2>>8));

        freq3=SID_freq_number (note, coarse3, fine3);
        sid.send(SID_FREQ3LO,char(freq3));
        sid.send(SID_FREQ3HI,char(freq3>>8));

        ctrl1|=1;
        sid.send(SID_CTRL1,ctrl1);
        ctrl2|=1;
        sid.send(SID_CTRL2,ctrl2);
        ctrl3|=1;
        sid.send(SID_CTRL3,ctrl3);
}

void synth_off (int note)
{
        ctrl1&=~1;
        sid.send(SID_CTRL1,ctrl1);
        ctrl2&=~1;
        sid.send(SID_CTRL2,ctrl2);
        ctrl3&=~1;
        sid.send(SID_CTRL3,ctrl3);
}
```

Listado 8-15. Funciones *synth_on* y *synth_off*

La función encargada de calcular el valor de los registros de frecuencia es *SID_freq_number*, que recibe como parámetros la nota y las variables de la afinación y devuelve el número de registro. Si la nota está fuera del rango de frecuencias del SID devuelve cero, y la voz no emitirá sonido. Esta misma función se usa también en el modo polifónico.

Para obtener la frecuencia correspondiente a cada nota se utiliza la escala de temperamento igual, que es la escala más comúnmente utilizada en la música occidental, y divide la octava en doce intervalos perceptualmente iguales, llamados semitonos. Esto se traduce en que la relación entre una nota y la siguiente es siempre la misma, y viene dada por la expresión (8.3). Puede observarse fácilmente que al subir doce semitonos (multiplicar doce veces por el factor) se obtiene el doble de frecuencia, es decir, la misma nota una octava por encima.

$$\frac{F_{note}}{F_{note-1}} = 2^{\frac{1}{12}} \tag{8.3}$$

En el sintetizador se guarda un array con los valores de los registros del SID para todas las notas de la octava más aguda que es capaz de interpretar el SID. Cuando se recibe una nota se calculan dos cosas: qué nota es (do, re, fa#, etc.), y por lo tanto la posición del array en que hay que buscarla, y en qué octava está. Sabiendo que la misma nota una octava más grave corresponde a la mitad de la frecuencia, y que dividir por dos un número digital es equivalente a desplazarlo un bit hacia la derecha, se obtiene fácilmente la frecuencia absoluta a interpretar. La especificación MIDI (MIDI Manufacturers Association, 1995) establece el número para cada nota. En la Tabla 8-2 se muestran los valores almacenados en el array de notas, junto con el número de nota MIDI y la frecuencia final obtenida, calculada con la expresión (3.1). La octava considerada comienza en el Si6 porque el Si7 está fuera del rango del oscilador, y no puede almacenarse con 16 bits.

Tabla 8-2. Valores almacenados en el array de notas, *ABS_NOTES [12]*

Nota	Si6	Do7	Do#7	Re7	Re#7	Mi7	Fa7	Fa#7	Sol7	Sol#7	La7	La#7
Frecuencia	1975,54	2093,02	2217,47	2349,32	2489,03	2637,03	2793,85	2959,97	3135,98	3322,42	3520,01	3729,28
Nota MIDI	95	96	97	98	99	100	101	102	103	104	105	106
Índice del array	0	1	2	3	4	5	6	7	8	9	10	11
Valor en SID	33144	35115	37203	39415	41759	44242	46873	49660	52613	55741	59056	62567

El proceso de cálculo se muestra en el Listado 8-16. El valor de la afinación gruesa, coarse, está comprendido entre 0 y 31, y para usarlo se le resta 16. Esto permite transponer cada voz hasta un máximo de 16 semitonos por debajo y 15 semitonos por encima. El valor de la afinación fina, *fine* es un número comprendido entre 0 y 127 (7 bits). Se pretende que este parámetro cubra todo el rango de un semitono, inalcanzable para la afinación gruesa. La aproximación utilizada es la interpolación lineal entre la frecuencia de la nota y la frecuencia de la nota siguiente, que siempre es un semitono más aguda, es decir, según la ecuación (8.4). Cabe mencionar que, aunque la octava más aguda del SID se suele numerar como la séptima, es la octava octava que puede manejar MIDI, y por eso el desplazamiento en octavas se calcula en relación a 8.

$$f_{salida} = f_{nota} + \frac{f_{nota+1} - f_{nota}}{128} \cdot fine \tag{8.4}$$

```
/*Variables:
        ABS_NOTES [12]     Array de notas, desde Si6 a La#7
*/
word SID_freq_number (byte note, byte coarse, byte fine)
{
        byte array_pos;                         //índice del array, 0-11
        byte note_octave;               //Octava de la nota. Cubre todo el rango MIDI, vale 0 para Do(-1)=8.18 Hz
        int trans;                              //nota MIDI transpuesta por coarse, puede ser negativo
        byte desp;                              //desplazamiento en octavas/bits
        word f_note,f_next, inc;                //frecuencia de ambas notas, e incremento a sumar a la nota "base"
        long temp;

        trans=note+coarse-16;           //número de la nota transpuesta por coarse.
        if (trans<0) {return 0;}

        array_pos = ((trans+1)%12);       //el array comienza en Si6
        note_octave =  ((trans+1)/12);
        if (note_octave>8) { return (0);} //si se excede el rango del SID
        desp = 8-note_octave;
        f_note = ABS_NOTES[array_pos]>>desp;

        //obtención de la frecuencia de la siguiente nota.
        array_pos = ((trans+2)%12);
        note_octave =  ((trans+2)/12);
        desp = 8-note_octave;
        f_next = ABS_NOTES[array_pos]>>desp;

        inc =((f_next-f_note)/128)*fine;

        temp=f_note+inc;
        if (temp <= 0xFFFF) return (word(temp));
        else return 0;
}
```

Listado 8-16. Cálculo del valor de los registros de frecuencia del SID, función *SID_freq_number*

8.7 Note on y note off en modo polifónico

Cuando el modo de funcionamiento es polifónico el comportamiento es un poco más complejo. Al recibir un mensaje *note on* con velocidad distinta de cero, se busca una voz que no esté activa, es decir, que no esté ninguna nota en ese momento. Cuando se encuentra, se toca la nota recibida en dicha voz. Para saber si una voz está libre se utiliza un array (*voice_note[3]*) que contiene la nota de los tres osciladores. Cuando la nota de un oscilador tiene el valor FF_H se sabe que está libre para tocar la nueva nota. Si no se encuentra ninguna voz libre, porque ya se están tocando tres notas, se guarda el número de nota en una variable llamada *last_note*. Si *last_note* tiene un valor válido, en cuanto se quede alguna voz libre se tocará *last_note* en dicha voz. El objeto de esto es dar algo más de facilidad al intérprete, no siendo demasiado riguroso con el momento en que se silencia una voz. Cuando se recibe un mensaje *note off* o *note on* con velocidad cero se busca en qué voz está sonando dicha nota. Si se encuentra dicho oscilador pueden darse dos circunstancias: no hay ninguna nota pendiente (*last_note* = FF_H), en cuyo caso simplemente se silencia la voz; o *last_note* tiene un valor válido, y se toca *last_note* en la voz encontrada. Si la nota recibida es la nota pendiente, es decir, *last_note*, se le asigna el valor FF_H, y ya no hay nota pendiente.

```
/*Variables
        freq1,freq2,freq3       valor de los registros de frecuencia de cada voz
        voice_note[3]           valor de la nota interpretada por cada oscilador
        last_note               nota pendiente
        ctrl_poly               registro de control común de las tres voces. Los cuatro bits bajos siempre valen cero.
        coarse_poly, fine_poly          controles de afinación comunes a las tres voces
        gate_poly               bit gate de cada voz, el valor de cada bit:  | 0 | 0 | 0 | 0 | GATE3 | GATE2 | GATE1 |
*/
if (voice_note[0]==note)//si la nota está tocándose en el oscilador 1
{
        if (last_note != 0xFF)  {       //hay nota pendiente, reemplazar la nota
                freq1=SID_freq_number (last_note, coarse_poly, fine_poly);
                sid.send(SID_CTRL1, ctrl_poly);
                sid.send(SID_FREQ1LO,char(freq1));
                sid.send(SID_FREQ1HI,char(freq1>>8));
                sid.send(SID_CTRL1, ctrl_poly | 1);
                voice_note[0]=last_note;
                last_note=0xFF;
        }
        else//no hay nota pendiente, silenciar oscilador.
         {
                sid.send (SID_CTRL1, ctrl_poly);
                voice_note[0]=0xFF;
                gate_poly &= ~1;
        }
}//osc.1
else if (voice_note[1]==note)//si la nota está tocándose en el oscilador 2
{
        if (last_note != 0xFF)
        {//hay nota pendiente, reemplazar la nota
                freq2=SID_freq_number (last_note, coarse_poly, fine_poly);
                sid.send(SID_CTRL2,ctrl_poly);
                sid.send(SID_FREQ2LO,char(freq2));
                sid.send(SID_FREQ2HI,char(freq2>>8));
                sid.send(SID_CTRL2,ctrl_poly | 1 );
                voice_note[1]=last_note;
                last_note=0xFF;
        }
        else//no hay nota pendiente, silenciar oscilador.
        {
                sid.send (SID_CTRL2, ctrl_poly);
                voice_note[1]=0xFF;
                gate_poly &= ~2;
        }
}//osc.2
else if (voice_note[2]==note)//si la nota está tocándose en el oscilador 3
{
        if (last_note != 0xFF)
        {//hay nota pendiente, reemplazar la nota
                freq3=SID_freq_number (last_note, coarse_poly, fine_poly);
                sid.send(SID_CTRL3, ctrl_poly);
                sid.send(SID_FREQ3LO,char(freq3));
                sid.send(SID_FREQ3HI,char(freq3>>8));
                sid.send(SID_CTRL3,ctrl_poly | 1);
                voice_note[2]=last_note;
                last_note=0xFF;
        }
        else//no hay nota pendiente, silenciar oscilador.
        {
                sid.send (SID_CTRL3, ctrl_poly);
                voice_note[2]=0xFF;
                gate_poly &= ~4;
        }
}//osc.3
else if (last_note=note) last_note=0xFF;
```

Listado 8-17. Procesado de *note off* y *note on* con velocidad cero para el modo polifónico

El registro de control, así como los parámetros de afinación, es común a todas las voces, y las modulaciones entre voces no se permiten en el modo polifónico. Como sus cuatro bits bajos (incluido

GATE) son siempre cero, escribirlo directamente equivale a silenciar la voz. La variable *gate_poly* sirve para tener almacenado el bit *GATE* de cada voz.

```
if (voice_note[0]==0xFF)//si el oscilador 1 está libre...
{
        freq1=SID_freq_number (note, coarse_poly, fine_poly);
        if (gate_poly & 1) sid.send(SID_CTRL1,ctrl_poly);
        gate_poly |= 1;                                    //activar GATE
        sid.send(SID_FREQ1LO,char(freq1));
        sid.send(SID_FREQ1HI,char(freq1>>8));
        sid.send(SID_CTRL1,ctrl_poly | 1);
        voice_note[0]=note;
}
else if (voice_note[1]==0xFF)//si el oscilador 2 está libre…
{
        freq2=SID_freq_number (note, coarse_poly, fine_poly);
        if (gate_poly & 2) sid.send(SID_CTRL2,ctrl_poly);
        gate_poly |= 2;                                    //activar GATE
        sid.send(SID_FREQ2LO,char(freq2));
        sid.send(SID_FREQ2HI,char(freq2>>8));
        sid.send(SID_CTRL2,ctrl_poly | 1);
        voice_note[1]=note;
}
else if (voice_note[2]==0xFF)//si el oscilador 3 está libre…
{
        freq3=SID_freq_number (note, coarse_poly, fine_poly);
        if (gate_poly &4) sid.send(SID_CTRL3,ctrl_poly);
        gate_poly |= 4;                                    //activar GATE
        sid.send(SID_FREQ3LO,char(freq3));
        sid.send(SID_FREQ3HI,char(freq3>>8));
        sid.send(SID_CTRL3,ctrl_poly | 1);
        voice_note[2]=note;
}
else    last_note=note;
```

Listado 8-18. Procesado de *note on* con velocidad distinta de cero en modo polifónico

8.8 Control Change

Los mensajes *control change* se utilizan en este caso para modificar los parámetros del sintetizador. Un mensaje control change tiene dos parámetros (bits de datos): número de controlador y valor del control. Lo que se hace es asignar un número de controlador a cada parámetro del sintetizador y modificarlo en función del valor recibido. En el caso del sintetizador podrían considerarse tres tipos de parámetros modificables, que se han llamado así:

- Binarios: sólo pueden tomar dos valores, 0 y 1. Se considera 1 cuando el valor del controlador recibido es mayor o igual que 64, y 0 si es menor. Por ejemplo el filtrado de una voz, o activar la modulación en anillo de una voz.
- "Continuos": El parámetro es un número que puede tomar todos los valores desde cero hasta el máximo. Por ejemplo la anchura de pulso de la onda, o la frecuencia de corte del filtro.

- Discretos: no son parámetros numéricos, y pueden tomar un conjunto determinado de valores. Es el caso de la selección de forma de onda, que sólo permite las cuatro formas básicas. En este caso hay que hacer una asignación manual entre valores del control y del parámetro.

En la Tabla 8-3, donde se puede consultar el número de controlador asignado a cada parámetro del sintetizador, así como la forma en que se usa el valor del control. En la función *doHandleControlChange* según el número de controlador recibido se procesa el valor del control de una forma u otra, usando una estructura *switch...case*.

Los parámetros número 7 y del 10 al 14 afectan al funcionamiento del filtro. Los parámetros 16 a 59 son característicos del modo monofónico, y sólo afectan a una de las voces. Si el modo seleccionado es el polifónico cuando se modifica uno de estos parámetros, se actualiza la variable asociada (por ejemplo *ctrl1* para el parámetro *RING1*), pero el cambio no repercute en los registros del SID. Así, cuando se seleccione el modo monofónico de nuevo el parámetro estará actualizado.

Los parámetros 64 a 72 son característicos del modo polifónico, y afectan a todas las voces por igual. Si el modo seleccionado es el monofónico cuando se modifica uno de estos parámetros, se actualiza la variable asociada (por ejemplo *ctrl_poly* para el parámetro *WAVE_POLY*), pero el cambio no repercute en los registros del SID.

Los parámetros 7 y del 10 al 14 actualizan la variable asociada al modo en que está el sintetizador cuando se modifican, y el cambio repercute en los registros del SID. Por ejemplo, cuando en modo polifónico se modifica el parámetro *MASTER*, se actualiza la variable *mode_poly* (que es una copia del registro *Mode/Vol*) y su valor se actualiza también en el registro del SID. Si el mismo parámetro se modifica en el modo polifónico se actualiza la variable *mode*, y se escribe en el SID.

El valor de un *control change* siempre es un número de 7 bits (el MSB de un byte de datos siempre es cero), pero algunos parámetros continuos tienen más o menos resolución. Cuando el parámetro en cuestión tenga más bits (por ejemplo la anchura de pulso), hay que desplazar el valor del control unos bits hacia la izquierda, y no se aprovecha toda la resolución que permite ese parámetro. Cuando el parámetro tiene menos bits (nivel de sostenimiento) hay que despreciar algunos de los bits más bajos.

Tabla 8-3. Mapeado de los parámetros del sintetizador en mensajes *control change*

NÚMERO CC	PARÁMETRO	TIPO	COMENTARIOS
7	MASTER	continuo	Volumen general; 4bits
8	INPUT	sin usar	-
9	MODE	binario	1 poly; 0 mono; Equivalente a CM 126, 127
10	LP	binario	1 filtro paso bajo on; 0 filtro paso bajo off;
11	BP	binario	1 filtro paso banda on; 0 filtro paso banda off
12	CUT	continuo	frecuencia de corte del filtro; 11bits
13	RES	continuo	resonancia del filtro; 4bits
14	HP	binario	1 filtro paso alto on; 0 filtro paso alto off;
15	TUNE	sin usar	-
16	WAVE1	discreto	0X none; 1X TRI; 2X SAW; 3X SQR; +4X NOISE
17	COARSE1	continuo	5 bits
18	FINE1	continuo	
19	DUTY1	continuo	Anchura del pulso; 12bits.
20	RING1	binario	1 modulación en anillo on; 0 off
21	SYNC1	binario	1 sync on; 0 off
22	ATACK1	continuo	4bits
23	DECAY1	continuo	4bits
24	SUSTAIN1	continuo	4bits
25	RELEASE1	continuo	4bits
26	OUT1	binario	1 filtrar; 0 salida directa
32	WAVE2	discreto	0X none; 1X TRI; 2X SAW; 3X SQR; +4X NOISE
33	COARSE2	continuo	5 bits
34	FINE2	continuo	
35	DUTY2	continuo	12bits.
36	RING2	binario	1 ring modulation on; 0 off
37	SYNC2	binario	1 sync on; 0 off
38	ATACK2	continuo	4bits
39	DECAY2	continuo	4bits
40	SUSTAIN2	continuo	4bits
41	RELEASE2	continuo	4bits
42	OUT2	binario	1 filtrar; 0 salida directa
48	WAVE3	discreto	0X none; 1X TRI; 2X SAW; 3X SQR; +4X NOISE
49	COARSE3	continuo	5 bits
50	FINE3	continuo	
51	DUTY3	continuo	12bits.
52	RING3	binario	1 on; 0 off
53	SYNC3	binario	1 on; 0 off
54	ATACK3	continuo	4bits
55	DECAY3	continuo	4bits
56	SUSTAIN3	continuo	4bits
57	RELEASE3	continuo	4bits
58	OUT3	binario	1 filtrar; 0 salida directa
59	3OFF	binario	desactiva la salida del osc.3
64	WAVE_POLY	discreto	0X none; 1X TRI; 2X SAW; 3X SQR; +4X NOISE
65	COARSE_POLY	continuo	5 bits
66	FINE_POLY	continuo	
67	DUTY_POLY	continuo	12bits.
68	ATTACK_POLY	continuo	4bits
69	DECAY_POLY	continuo	4bits
70	SUSTAIN_POLY	continuo	4bits
71	RELEASE_POLY	continuo	4bits
72	OUT_POLY	binario	1 filtrar; 0 salida directa (todas las voces)
73	SAVE_PATCH	continuo	Guarda todos los parámetros en el patch indicado
74	LOAD_PATCH	continuo	Carga todos los parámetros del patch indicado
126	POLY OFF	ch. mode	Independiente del valor. Activa el modo monofónico.
127	POLY ON	ch. mode	Independiente del valor. Activa el modo polifónico.

Los mensajes de modo de canal (*channel mode messages*) son un tipo diferente de mensajes MIDI, que llevan información sobre cómo responderá el dispositivo a los mensajes de canal. El byte de estado es el mismo que el de un *control change*, y por lo tanto se pueden tratar con la misma función. El sintetizador responde a los mensajes de modo de canal POLY_OFF y POLY_ON. En el Listado 8-19 se muestra el fragmento de la función *DoHandleControlChange* que se encarga de esto. Actualizar el modo mediante dos tipos de mensajes es redundante, pero se conserva el controlador 9 porque se contempla que en futuras mejoras se implementen más modos de funcionamiento además de simplemente monofónico o polifónico.

```
void DoHandleControlChange (byte channel, byte number, byte value)
{
        switch (number) {
                case CC_MODE:                   //9
                        if (value>63) setSynthMode(MODE_POLY);
                        else setSynthMode(MODE_MONO);
                        break;

                case CM_POLY_ON:                        //127
                        setSynthMode(MODE_POLY);
                        break;

                case CM_POLY_OFF:                       //126
                        setSynthMode(MODE_MONO);
                        break;
/*****************************************************
                OTROS PARÁMETROS...
*****************************************************/
        }
}
```

Listado 8-19. Actualización de parámetros: modo de funcionamiento del sintetizador

La función *setSynthMode*, que cambia el modo de funcionamiento del sintetizador se muestra en el Listado 8-20. Al cambiar de modo se silencian todas las voces, se actualizan la variale *synth_mode* y se vuelcan los registros del SID mediante la función *synth_init*. En el Listado 8-21 se muestra, a modo de ejemplo de actualización de parámetros continuos, el fragmento de *DoHandleControlChange* que gestiona el cambio de los tiempos de ataque de la voz 1. Como ambos parámetros están expresados con 4 bits, el proceso es despreciar los 3 bits bajos del valor del control. Además el ataque se almacena en la parte alta del registro, por lo que hay que desplazar el valor un bit a la derecha (el MSB del valor es 0) e ignorar los cuatro bits bajos.

Obsérvese que cuando se modifica un parámetro de la voz 1 la variable asociada se actualiza, pero, al ser parámetros propios del modo monofónico, el cambio sólo repercute en el SID si el sintetizador funciona en dicho modo.

```
void setSynthMode (byte new_mode)
{
	if (synth_mode != new_mode)
	{
		if (new_mode==MODE_POLY)
		{
			synth_mode=MODE_POLY;
			synth_off(1);		//silenciar voces...
			last_note=0xFF;
			current_note=0xFF;
			voice_note[0] =0xFF;
			voice_note[1] =0xFF;
			voice_note[2] =0xFF;
			gate_poly = 0;

			synth_init();		//volcar variables en el SID
		}
		else if (new_mode == MODE_MONO)
		{
			synth_mode=MODE_MONO;
			synth_off(1);
			current_note=0xFF;

			synth_init();		//volcar variables en el SID
		}
}
```

Listado 8-20. Función setSynthMode, que modifica el modo de funcionamiento del sintetizador

```
void DoHandleControlChange (byte channel, byte number, byte value)
{
	switch (number) {
/*****************************************************
		OTROS PARÁMETROS...
*****************************************************/

		case CC_ATTACK1:
			ad1 &= 0x0F;
			value = (value<<1)&0xF0;
			ad1 |= value;
			if (synth_mode==MODE_MONO)
			{
				sid.send(SID_AD1,ad1);
			}
			break;

		case CC_DECAY1:
			ad1 &= 0xF0;
			ad1 |= (value>>3);
			if (synth_mode==MODE_MONO)
			{
				sid.send(SID_AD1,ad1);
			}
			break;

/*****************************************************
		OTROS PARÁMETROS...
*****************************************************/
	}
}
```

Listado 8-21. Actualización de parámetros. Modificación de los tiempos de ataque y caída de la voz 1

La forma de onda de las voces es un parámetro discreto, es decir, sólo puede tomar determinados valores no numéricos. La modificación de dicho parámetro se ejemplifica en el Listado 8-22. Se consideran

solamente los cuatro bits altos del valor recibido, y en función de ellos se actualiza el registro de control. Nótese que se permite no seleccionar ninguna forma de onda, repercutiendo en que la voz no suene en la salida, pero no se permiten formas de onda compuestas. En el Listado 8-23 se muestra la modificación de las modulaciones de una voz, como ejemplo de actualización de parámetros binarios.

```
void DoHandleControlChange (byte channel, byte number, byte value)
{
        switch (number) {
/*****************************************************
                OTROS PARÁMETROS...
*****************************************************/

        case CC_WAVE1:
                value= value >>4;
                switch (value) {
                        case 0:
                        value = 0;              //Ninguna forma de onda
                        break;
                        case 1:
                        value = 16;                     //triangular
                        break;
                        case 2:
                        value = 32;                     //diente de sierra
                        break;
                        case 3:
                        value = 64;                     //pulso
                        break;
                        default:
                        value = 128;            //ruido
                        break;
                        }
                        ctrl1&=0x0F;
                        ctrl1|=value;           //se elimina la forma de onda anterior y se actualiza.
                        if (synth_mode==MODE_MONO)
                        {
                                sid.send(SID_CTRL1,ctrl1);
                        }
                break;
/*****************************************************
                OTROS PARÁMETROS...
*****************************************************/
        }
}
```

Listado 8-22. Actualización de parámetros. Modificación de la forma de onda de la voz 1.

Por último, el guardado/cargado de patches, que también se realiza mediante mensajes control change, se muestra en el Listado 8-24. Se utilizan las funciones *savePatch* y *loadPatch*, que se explican en el apartado 8.9. Dichas funciones reciben como parámetro el número de patch a guardar/cargar, y dicho parámetro es directamente el valor del controlador recibido. Actualmente el número de patch puede estar entre 0 y 7, y si el número está fuera del rango, sencillamente se ignora.

```
void DoHandleControlChange (byte channel, byte number, byte value)
{
        switch (number) {
/*********************************************************
                OTROS PARÁMETROS...
**********************************************/

                case CC_SYNC1:
                        if (value>63) {
                                ctrl1|=2;
                        }
                        else {
                                ctrl1&=~2;
                        }
                        if (synth_mode==MODE_MONO) sid.send(SID_CTRL1,ctrl1);
                break;

                case CC_RING1:
                        if (value>63) {
                                ctrl1|=4;
                        }
                        else {
                                ctrl1&=~4;
                        }
                        if (synth_mode==MODE_MONO) sid.send(SID_CTRL1,ctrl1);
                        break;
/*********************************************************
                OTROS PARÁMETROS...
**********************************************/
        }
}
```

Listado 8-23. Actualización de parámetros. Activado/desactivado de las modulaciones de la voz 1.

```
 void DoHandleControlChange (byte channel, byte number, byte value)
{
        switch (number) {
/*********************************************************
                OTROS PARÁMETROS...
**********************************************/

        case CC_SAVE_PRESET:
                savePatch(value);
                break;

        case CC_LOAD_PRESET:
                loadPatch(value);
                break;
/*********************************************************
                OTROS PARÁMETROS...
**********************************************/
        }
}
```

Listado 8-24. Actualización de parámetros. Guardado/cargado de patches

8.9 Soporte de presets

El soporte de presets se ha implementado en la memoria EEPROM integrada en el Atmega 168, ya que, a diferencia de la RAM, no se borra al desconectar la alimentación. En un *patch* (preset) se almacenan

todos los parámetros necesarios para tener la misma configuración del sonido del sintetizador, esto es, todos los registros del SID y los controles de afinación (por duplicado, monofónico y polifónico) y el modo de funcionamiento. En este momento el sintetizador puede almacenar 8 presets, y cada uno ocupa 40 bytes. El mapa de la memoria EEPROM utilizada se muestra en la Tabla 8-4. Cada dirección de memoria tiene el tamaño de un byte.

Tabla 8-4. Mapa de memoria EEPROM

Dir. comienzo (DEC)	Dir. Final (DEC)	Info. almacenada
0	12	HEADER
13	52	PATCH0
53	92	PATCH1
93	132	PATCH2
133	172	PATCH3
173	212	PATCH4
213	252	PATCH5
253	292	PATCH6
293	332	PATCH7

En las primeras direcciones de la memoria se almacena una cabecera de 12 bytes, que contiene una cadena de caracteres que se muestra en la Tabla 8-5. En esta cadena, el número corresponde a la versión del programa con que se guardaron los presets. Cada vez que se cargue o guarde un *patch*, antes se lee esta cabecera, para comprobar que la memoria soporta *presets*. Sólo si la cabecera coincide con la esperada se continúa con la operación.

Tabla 8-5. Cabecera de la memoria de presets.

Dirección	0	1	2	3	4	5	6	7	8	9	10	11	12
Valor	‘S’	’I’	’D’	’u’	’i’	’n’	’a’	’s’	’t’	‘e’	‘r’	‘1’	(vacío)

El contenido de cada patch se puede ver en la Tabla 8-6. En esta tabla, el índice es la “dirección relativa” en que está almacenado el dato. La dirección absoluta se obtiene sumando el índice a la dirección de comienzo del patch en cuestión. Los tres primeros datos almacenados en el *patch* son tres caracteres correspondientes al nombre del *patch*. Es una función que no se ha implementado, pero se ha contemplado como posibilidad para futuras modificaciones. El valor de *synth_mode*, como ya se ha comentado es el modo de funcionamiento del sintetizador, y puede valer las constantes MODE_MONO (1) o MODE_POLY (2). Si tiene cualquier otro valor se considera que el *patch* está vacío.

Tabla 8-6. Estructura de un patch

	Índice	Parámetro
Info. Patch	0	Nom1
	1	Nom2
	2	Nom3
	3	synth_mode
VOZ 1 (MONO)	4	ctrl1
	5	coarse1
	6	fine1
	7	pw1 (lo)
	8	pw1 (hi)
	9	ad1
	10	sr1
VOZ 2 (MONO)	11	ctrl2
	12	coarse2
	13	fine2
	14	pw2 (lo)
	15	pw2 (hi)
	16	ad2
	17	sr2
VOZ 3 (MONO)	18	ctrl3
	19	coarse3
	20	fine3
	21	pw3 (lo)
	22	pw3 (hi)
	23	ad3
	24	sr3
VOCES POLY	25	ctrl_poly
	26	coarse_poly
	27	fine_poly
	28	pw_poly (lo)
	29	pw_poly (hi)
	30	ad_poly
	31	sr_poly
FILTRO MONO	32	freq (lo)
	33	freq (hi)
	34	mode
	35	res
FILTRO POLY	36	freq_poly (lo)
	37	freq_poly (hi)
	38	mode_poly
	39	res_poly

Para acceder a la memoria EEPROM del Atmega se utiliza la librería EEPROM.h, que se descarga automáticamente con la IDE de Arduino, y que se debe incluir al principio del programa mediante la directiva *#include <EEPROM.h>*. Básicamente se utilizan dos métodos de esta librería:

- ***EEPROM.write (dirección, dato)***, que escribe almacena el byte (dato) en la dirección indicada.
- ***EEPROM.read (dirección)***, que devuelve el byte almacenado en la dirección indicada.

En el Listado 8-26 se muestra la función resumida que guarda un preset en memoria. Se ha sobrecargado la función porque se ha considerado que para futuras modificaciones se puedan almacenar presets de un solo modo de funcionamiento, ahorrando memoria, o permitiendo cargar el modo monofónico de un patch y el modo polifónico de otro, aunque no se ha implementado. Así, al simplemente pasarle a la función como parámetro el número de preset por defecto se guardan todas las variables de los dos modos. En cuanto a la función *loadPatch*, que se muestra en el Listado 8-27, primero se cargan los valores almacenados en las variables del sintetizador, y luego se vuelcan en el SID mediante la función *synth_init*.

Por supuesto, antes de poder utilizar la memoria EEPROM para almacenar presets hay que inicializarla, es decir, escribir la cabecera. Para eso se utiliza la función *init_EEPROM*, que sólo es necesario ejecutar una vez. Primero, en la función *setup* del programa *SIDuinaster.ino*, se incluye en algún punto la llamada a esta función, da igual en qué parte. El código se sube a la placa mediante la instrucción *FILE->Upload*. Así, el programa se compila y se transmite al microcontrolador de la placa Arduino, y una vez terminado el proceso, el programa se ejecuta, y con él la función *init_EEPROM*, que se muestra en el Listado 8-25. Sin necesidad de desconectar la placa del ordenador, se quita la línea de código que llamaba a esta función y se vuelve a subir el código a la placa. De este modo, la EEPROM queda inicializada para el soporte de presets, pero no se inicializa cada vez que se conecta el sintetizador.

```
/************************************************
LLamada la primera vez, en la función setup:
setup ()
{
        /************************************************
                RESTO DE LA FUNCIÓN  SETUP.....
        ************************************************/
        init_EEPROM ();
}
************************************************/

void init_EEPROM()
{
        byte i;
        char* header = HEADER_STRING;
        for (i=0; header[i] != NULL; i++) EEPROM.write (i, header[i]);
}
```

Listado 8-25. Función *init_EEPROM*, que inicializa la EEPROM para almacenamiento de presets

```
/*Constantes:
        NUMBER_OF_PATCHES       8
        MODE_ALL            0xFF            se almacena info. del modo monofónico y polifónico
        EEPROM_ADDRESS_OFFSET 13            la dirección del primer bit del patch0
        PATCH_SIZE          40
        HEADER_STRING   "SIDuinaster1"
*/
byte savePatch(byte number)
{//devuelve 1 si se ha guardad corectamente , 0 si no
        return (savePatch(number, MODE_ALL));  //por defecto guardar todo el preset
}

byte savePatch (byte number, byte mode_to_save)
{
        byte address, i;
        char* header=HEADER_STRING;         //esta es la cabecera que debería estar en las primeras direcciones

        if (number < NUMBER_OF_PATCHES)
        {
                for (i=0; header[i] != NULL; i++)           //comprobar la cabecera
                {
                        if (EEPROM.read(i) != header[i])            {return 0;}
                }
                if (mode_to_save==MODE_ALL)
                {
                        cli();      //desactivar interrupciones durante el proceso de escritura en la EEPROM
                        address= EEPROM_ADDRESS_OFFSET+(number*PATCH_SIZE);//start address
                        address+=3;//saltar a la direccion del primer dato valido (synth_mode)

                        EEPROM.write(address++,synth_mode);
                        EEPROM.write(address++,ctrl1&0xFE);//desactivar el bit GATE, no tiene sentido guardarlo
                        EEPROM.write(address++,coarse1);
                        EEPROM.write(address++,fine1);
                        EEPROM.write(address++,char(pw1));      //byte bajo
                        EEPROM.write(address++,char(pw1>>8)); //byte alto
                        EEPROM.write(address++,ad1);
                        EEPROM.write(address++,sr1);
/*****************************************************************************
                        OTROS PARÁMETROS….
*****************************************************************************/
                        EEPROM.write(address++,char(freq_poly));
                        EEPROM.write(address++,char(freq_poly>>8));
                        EEPROM.write(address++,mode_poly);
                        EEPROM.write(address++,res_poly);
                        sei();      //reactivar interrupciones
                }
        }//(number < NUMBER_OF_PATCHES)
        return 1;
}
```

Listado 8-26. Función *savePatch*, que guarda un preset en la memoria EEPROM

```
/*Constantes:
        NUMBER_OF_PATCHES       8
        MODE_ALL        0xFF            se almacena info. del modo monofónico y polifónico
        EEPROM_ADDRESS_OFFSET 13        la dirección del primer bit del patch0
        PATCH_SIZE      40
        HEADER_STRING   "SIDuinaster1"
*/
byte loadPatch (byte number)
{//devuelve 1 si se consigue cargar el patch, 0 si no
        return (loadPatch(number, MODE_ALL));//por defecto cargar todos los parámetros
}

byte loadPatch (byte number, byte mode_to_load)
{
        byte address, i;
        char* header=HEADER_STRING;

        if (number < NUMBER_OF_PATCHES)
        {//first check if EEPROM is correct:
                for (i=0; header[i] != NULL; i++)
                {
                        if (EEPROM.read(i) != header[i])        {return 0;}
                }

                if (mode_to_load==MODE_ALL)
                {
                        address= EEPROM_ADDRESS_OFFSET+(number*PATCH_SIZE);
                        address+=3;

                        i=EEPROM.read(address++);     //synth_mode, ha de ser un valor conocido
                        if ( (i != MODE_POLY) && ( i != MODE_MONO) ) {return 0;}

                        cli();
                        synth_mode=i;
                        ctrl1=EEPROM.read(address++);
                        coarse1=EEPROM.read(address++);
                        fine1=EEPROM.read(address++);
                        pw1=EEPROM.read(address++);
                        pw1|=(EEPROM.read(address++))<<8;
                        ad1=EEPROM.read(address++);
                        sr1=EEPROM.read(address++);
/*****************************************************************************
                        OTROS PARÁMETROS….
*****************************************************************************/
                        freq_poly=EEPROM.read(address++);
                        freq_poly|=(EEPROM.read(address++))<<8;
                        mode_poly=EEPROM.read(address++);
                        res_poly=EEPROM.read(address++);
                        synth_init ();
                        sei();
                }
        }
        return 1;
}
```

Listado 8-27. Función *loadPatch*, que carga un preset de la memoria EEPROM

9. SOFTWARE ADICIONAL

Además del sintetizador en sí, para poder controlarlo fácilmente desde el ordenador hace falta cierto software que envíe al sintetizador vía USB los mensajes *control change* necesarios para modificar sus parámetros. Para controlar el sintetizador, es necesario:

- Un software que genere los mensajes *control change* de MIDI. Es necesario que sea configurable y sencillo de manejar, para poder enviar fácilmente los mensajes específicos del sintetizador. Se ha utilizado un plugin VST (*Virtual Studio Technology*) llamado *midiPads,* que se puede descargar gratuitamente en el sitio web del diseñador (Insert Piz here). Al ser *midiPads* un plugin necesita ejecutarse desde un programa *Host*. Se ha utilizado un *host VST* llamado *SAVIHost.*
- Un software que convierta los mensajes MIDI que maneja el ordenador, y los envíe a través de USB para que los pueda recibir el sintetizador. Se ha utilizdo un programa llamado *Hairless MIDI-Serial bridge,* que se puede descargar en (Gratton, 2011).
- Otro software que conecte los dos programas anteriormente mencionados, que funciona como un panel de conexiones dentro del ordenador. Se utiliza para esto la aplicación *MIDI Yoke,* descargable en (O'Connell).

Todos estos programas son *freeware*, siguiendo la filosofía de crear un sistema de bajo coste. Han sido la opción escogida para este proyecto, pero si se puede disponer de otros programas que realicen las mismas funciones la solución es perfectamente válida.

9.1 MIDI Yoke

Para que el sistema pueda funcionar en primer lugar es necesario instalar *MIDI Yoke*. Crea una serie de puertos MIDI virtuales a los que pueden conectarse aplicaciones que usen MIDI. Se dice que son virtuales porque no corresponden a ninguna conexión física, pero los programas envían datos a esos puertos como si fuesen puertos MIDI reales.

MIDI Yoke se puede descargar gratuitamente del sitio web de MIDI Ox (O'Connell). La instalación puede dar algún problema en los sistemas operativos Windows Vista y Windows 7, en cuyo caso hay que desactivar temporalmente el control de cuentas de usuario en *Panel de control-> Sistema y seguridad-> Cambiar configuración de control de cuentas de usuario.* Una vez instalado, los programas que manejan MIDI tendrán a su disposición una serie de puertos MIDI, de la misma manera que un programa que maneja USB tiene acceso a los puertos USB.

MIDI Yoke sirve para comunicar distintos programas que manejen MIDI, proporcionando un puerto común. Funciona como un panel de conexiones, proporcionando una serie de entradas y salidas como se muestra en la Figura 9.1. El llamado programa 1 (que en el caso del sintetizador es *SAVIHost*) tiene una salida MIDI, que en el mismo programa se configura para estar conectada al puerto *MIDI Yoke 1*. A su

vez el programa 2 (que es *Hairless MIDI-Serial Bridge*) se elige que la entrada MIDI tome la información del puerto *MIDI Yoke 1*. La entrada de un puerto está automáticamente conectada a la salida, por lo que los mensajes MIDI generados por el programa 1 pasarán directamente a la entrada MIDI del programa 2.

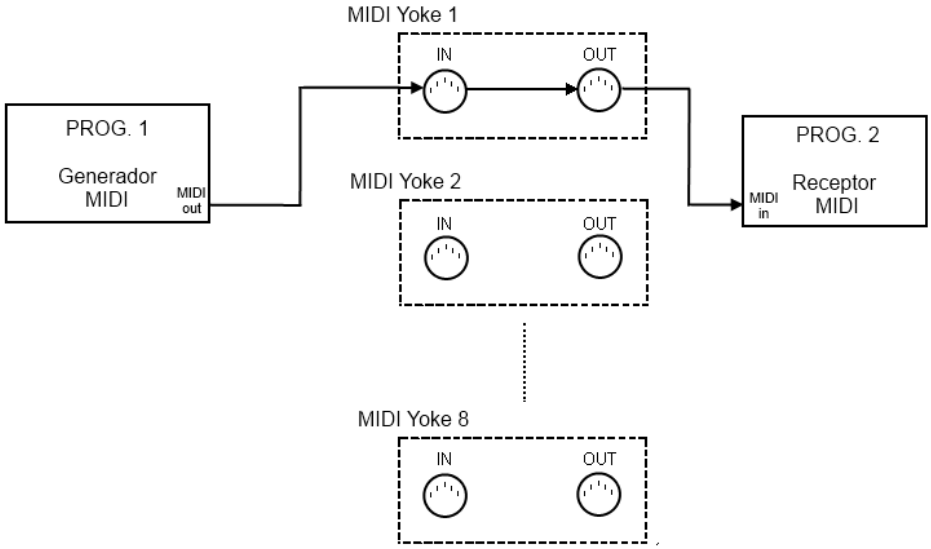

Figura 9.1. Esquema de una conexión MIDI genérica a través de *MIDI Yoke*

Hay que tener cuidado usando *MIDI Yoke*, pues muchos programas tienen a la vez entradas y salidas MIDI, y puede producirse lo que se llama realimentación MIDI (*MIDI feedback*), si se cierra un bucle y los mensajes generados vuelven a la entrada, pudiendo bloquear el sistema.

9.2 midiPads

midiPads es un plugin VST gratuito (más concretamente un instrumento VST), que proporciona una interfaz de usuario que muestra un panel configurable, en el que se pueden insertar botones/sliders para generar mensajes MIDI fácilmente. Entre otras cosas, se pueden insertar en el panel:

- Botones que al pulsarlos generen un mensaje *control change* con un número de controlador y un valor de control determinados, configurables por el usuario.
- Botones de tipo on/off, que al pulsarlos una vez queneren un mensaje *control change* con un número de controlador y un valor de control determinados, y al pulsarlos otra vez generen el mismo mensaje con otro valor de control.
- *Sliders*, que al mover el control en el eje horizontal o vertical genere un valor de control cualquiera entre 0 y 127 para un número de controlador determinado.

Lo que se ha hecho es montar un panel con sliders para controlar los parámetros “continuos” y botones para controlar los parámetros discretos y binarios del sintetizador, y se ha guardado dicha configuración en un banco de programa llamado *SIDtetizador*. Como *midiPads* es un *plugin*, no puede funcionar por sí

mismo, sino que depende siempre de otro programa, un *host VST*. Los programas Nuendo o Cubase de Steinberg aceptan plugins VST, pero existen otras opciones más simples, además de gratuitas para ejectutar plugins VST. En este caso se utiliza un programa freeware llamado *SAVIHost*, que sirve para ejecutar rápidamente y de forma sencilla un único plugin VST.

Se puede descargar *midiPads* gratuitamente en el sitio web del diseñador (Insert Piz here). Para que funcione, en la carpeta *midiPads\midiPadBanks* debe estar el archivo *SIDtetizador.mpadb*, que contiene la configuración del panel para configurar el SID. Además para que se puedan ver los iconos de los botones la carpeta *Iconos SIDtetizador* debe estar en el directorio *midiPads\midiPadIcons.*

El procedimiento para poner a funcionar (en Windows) *midiPads* con *SAVIHost* es:

- Ejecutar el fichero *midiPads.exe*, contenido en la carpeta MIDIPADS que se incluye en el CD de este proyecto. Este fichero es una copia renombrada el archivo ejecutable *savihost.exe*, que al estar en la misma carpeta en que está el plugin midiPads (*midiPads.dll)*, al ejecutarlo directamente se abre el plugin.
- En el botón *menu* de *midiPads*, seleccionar *Save/Load->Load Bank/Patch* y cargar *midiPads\midiPadBanks\SIDtetizador.mpadb*. El panel de control del SID debe aparecer en la ventana del plugin, y tiene el aspecto de la Figura 9.2.
- A continuación encaminar la salida MIDI hacia el puerto de *MIDI Yoke:* En la barra de menús de *SAVIHost*, abrir *devices->MIDI...* seleccionar como output port *MIDI Yoke 1*.

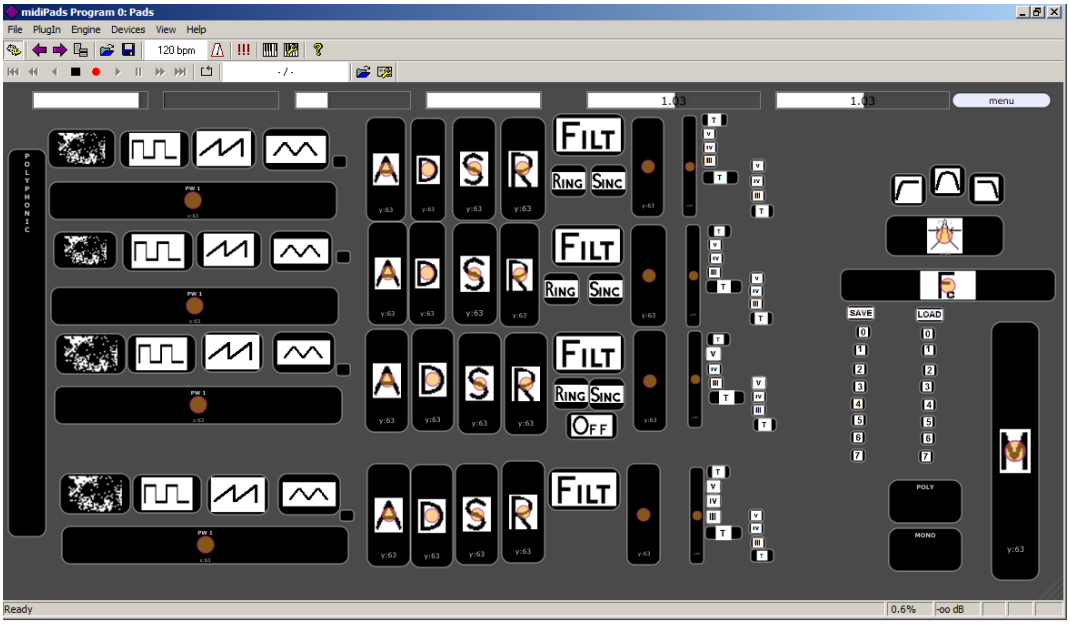

Figura 9.2. Panel de control del SID en midiPads

Todos los mensajes *control change* generados por *midiPads* al pulsar sus botones irán a parar al puerto *MIDI Yoke 1*, al que estará conectado el conversor MIDI-USB. Los números de controlador y valor de control generados por cada botón son los de la Tabla 8-3.

En cuanto al panel de control del sintetizador, a la izquierda están los botones que controlan todos los parámetros relativos a las voces. Se puede ver que hay cuatro filas con prácticamente los mismos botones, y corresponden, en orden de arriba abajo, a los parámetros de: Voz 1 (monofónico), Voz 2 (monofónico), Voz 3 (monofónico) y todas las voces (polifónico). A la izquierda de cada fila de voz están los selectores de forma de onda, y debajo el slider para controlar la anchura de pulso (nótese que es posible no seleccionar ninguna forma de onda, anulando el sonido de la voz).

Los siguientes sliders verticales corresponden a los parámetros de la curva ADSR, y justo a su derecha los botones *FILT* (que selecciona si la voz se filtra o pasa directamente a la salida) y *Ring* y *Sync* (que activan/desactivan las modulaciones). Estos son un tipo especial de botones (on/off), que al pulsar una vez se activan y mandan un valor del controlador en cuestión (127) y al volverlos a pulsar se desactivan y mandan otro valor (0).

Los dos sliders verticales justo a la derecha son para los parámetros *coarse* (el más grueso) y *fine* (el más fino) de cada voz. Los botones a la derecha de estos sliders controlan también la afinación gruesa, pero envía cada uno un valor concreto para transponer intervalo frecuentemente utilizado. En concreto en la columna de la izquierda se tienen los valores para transponer 0 semitonos (tónica, símbolo *T*), 4 semitonos (tercera mayor, *III*), 5 semitonos (cuarta justa, *IV*), 7 semitonos (quinta justa, *V*) y 12 semitonos (octava, *VIII*). En la columna de la derecha son los mismos intervalos pero una octava por debajo de la tónica.

Arriba a la derecha se controlan los parámetros del filtro: botones para activar/desactivar cada tipo de filtro, y sliders horizontales para controlar la resonancia y la frecuencia de corte. El slider de abajo a la derecha corresponde al volumen general del SID. Las dos columnas de botones pequeños a la derecha del panel se utilizan para guardar (izquierda) y cargar (derecha) los presets del sintetizador. Los botones de debajo sirven para cambiar entre el modo polifónico y el monofónico, usando los mensajes *channel mode* de *poly off* y *poly on*. Lo mismo hace el botón on/off de la izquierda del todo.

9.3 Hairless MIDI-Serial Bridge

El programa en cuestión básicamente tiene una entrada MIDI, una salida MIDI, y una entrada/salida serie. Todos los mensajes MIDI que reciba por el puerto serie los reproduce en su salida MIDI y todos los mensajes MIDI que reciba de su entrada MIDI los reproduce en el puerto Serie. Este programa es de código abierto, y se puede descargar del sitio web del diseñador (Gratton, 2011).

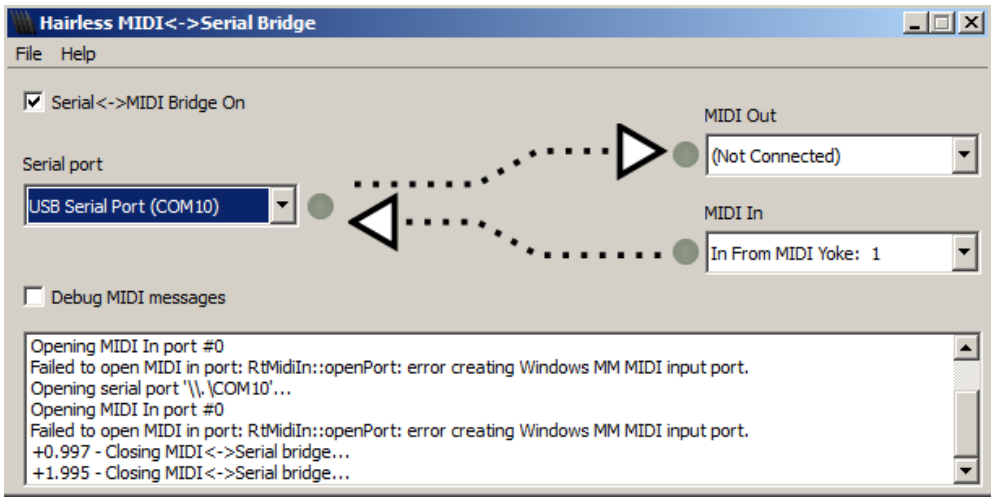

Figura 9.3. Ventana principal del programa *Hairless MIDI-Serial Bridge*

En la Figura 9.3 se muestra el aspecto de la ventana principal del programa conversor. *Hairless MIDI-Serial Bridge* no necesita instalación, se puede ejecutar directamente. La configuración del programa consiste en:

- Configurar el puerto serie: en *File->preferences* cambiar la velocidad de transmisión (*Baud rate*) a 57600 bit rate. Este valor ha de ser el mismo que el configurado en la placa de control. Para el resto de parámetros los valores por defecto son correctos.
- Conectar la entrada MIDI del programa a *MIDI Yoke 1*. De este modo, los mensajes MIDI generados por *midiPads* se transmitirán al puerto serie. La salida MIDI del programa no es necesario conectarla.
- Una vez conectado el sintetizador al ordenador por USB, en el puerto serie seleccionar el puerto al que esté conectado. Probablemente sea el único puerto que aparezca como posible.
- Activar el puente activando la opción *Serial<->MIDI Bridge On*. Con esto es suficiente para controlar el sintetizador desde el ordenador.

10. PRESUPUESTO

A continuación se detalla el presupuesto del proyecto:

Material de trabajo:

	Precio (€)/unidad	Cantidad	Precio
Ordenador portátil	400	1	400
Cable USB	10	1	10
Herramientas soldadura	35	1	35
		Total:	**445**

Material para la construcción del prototipo:

		Precio (€)/unidad	Cantidad	Precio	
Placa de control	Arduino Duemilanove	21	1	21	
Placa de generación de sonido, versión "hardware"	Placa de circuito impreso	4,70	1	4,70	
	SID	35,95	1	35,95	
	LT-1073	6,42	1	6,42	
	74HC595	0,57	2	1,14	
	Pulsador	0,30	1	0,30	
	Conector jack 3,5 mm	0,25	1	0,25	
	Condensador tántalo	0,20	4	0,80	
	Bobina 150 uH	0,58	1	0,58	
	Zócalo DIP-8	0,29	1	0,29	
	Zócalo DIP-16	0,58	2	1,16	
	Zócalo DIP-28 grueso	1,13	1	1,13	
	Tira 40 pin hembra cuadrado	0,41	1	0,41	
	Tira 40 pin macho cuadrado largo	0,66	1	0,66	**Subtotal:**
	Otros componentes	0,25	1	0,25	**54,04**
Placa de generación de sonido, versión "software"	Placa de circuito impreso	4,70	1	4,70	
	microprocesador Atmega 168P	5,08	1	5,08	
	Cristal 16 MHz	0,21	1	0,21	
	Pulsador	0,30	1	0,30	
	Conector jack 3,5 mm	0,25	1	0,25	
	Zócalo DIP-28	1,03	1	1,03	
	Pin hembra cuadrado 40 pin	0,41	1	0,41	
	Pin macho cuadrado largo 40 pin	0,66	1	0,66	**Subtotal:**
	Otros componentes	0,12	1	0,12	**12,76**
			Total*:	**87,79**	

Horas de trabajo:

	Precio (€)/hora	Horas	Precio (€)
Estudios previos	30	50	1500
Diseño, codificación y pruebas	30	210	6300
Construcción	30	10	300
		Total:	**8100**

COSTE TOTAL DEL PROYECTO (€):	**8632,79**

*La placa ArdoMIDI utilizada para la recepción MIDI ha sido prestada por el tutor del proyecto, Lino García, y por ello no se incluye en el presupuesto.

11. CONCLUSIONES

El objetivo del proyecto se ha cubierto: se ha construido un sintetizador de bajo coste basado en el SID (versión software y hardware), controlable vía MIDI. Además es sencillo de construir por cualquier persona, montando las placas de circuito impreso de cada módulo y siguiendo las instrucciones para subir los códigos (en el CD del proyecto se incluyen los layouts de las PCB). Sin embargo, como a continuación se indica, hay algunas consideraciones que deben hacerse sobre el resultado, tanto de defectos a ser resueltos como de ampliaciones de la funcionalidad.

11.1 Comparación SID-emulador

La versión software del sintetizador, frente al circuito del SID original, tiene la ventaja de su reducido precio (recuérdese que uno de los objetivos del proyecto era el bajo coste). Además, previsiblemente, el precio del SID aumente con el tiempo, debido a su escasez creciente.

Otro problema es la calidad de los SIDs del mercado de segunda mano. Es muy difícil encontrar chips en buen estado después de tantos años sin fabricarse. En los chips que se han manejado, el filtro no funcionaba, por lo que no se ha podido hacer uso de toda la funcionalidad del chip. Además, cuando los chips se calentaban, los moduladores de amplitud de las voces dejaban de funcionar correctamente, no silenciando las voces cuando se ponía a cero el bit GATE. Este es un problema que no tiene solución, salvo que se considere como tal mejorar el emulador software para igualar las características del SID. A pesar de todo esto, la calidad del sintetizador en versión hardware (siempre que no se usen los filtros) tiene una calidad bastante buena.

Sin embargo, el emulador no es lo suficientemente fiel al SID. Sus principales defectos son la calidad de audio, y las funciones del SID no implementadas. La calidad de audio es muy inferior a la del SID, debido a la resolución de la muestra de audio y la modulación PWM. La modulación PWM como conversión D/A puede dar buen resultado si la frecuencia de PWM es muy superior a la frecuencia de muestreo, pero en el emulador no es posible esto porque el mismo módulo contador del Atmega se utiliza para temporizar el cálculo de la muestra de audio y para la modulación PWM. Esto es porque el mismo módulo contador del Atmega168 se utiliza para temporizar el cálculo de la muestra de audio y para la modulación PWM.

La resolución de audio del emulador es de 8 bits, inferior a la del SID, que utiliza 12 bits. Esto, sumado al efecto de la baja frecuencia de PWM disminuye considerablemente la calidad de audio. No es solución aumentar la resolución porque por cada bit que se incrementara, la frecuencia de PWM habría de reducirse a la mitad, y ésta ya es suficientemente baja. Las funciones no implementadas en el emulador son el filtro, el control de volumen y la modulación *hard sync*, reduciendo las posibilidades del emulador con respecto a las del SID.

11.2 Evaluación general del sintetizador

En general el sintetizador, aún siendo mejorable en su funcionalidad, funciona satisfactoriamente, no obstante cabe hacer al menos una consideración. En cuanto a la lógica del sintetizador, es muy mejorable el control de la afinación fina, que cubre el rango de un semitono. La aproximación lineal supone calcular las frecuencias intermedias entre una nota y la siguiente como una línea recta, mientras que se sabe que la percepción humana se aproxima más a una curva exponencial. Esta aproximación no es suficientemente buena y hace que el sintetizador pueda sonar desafinado al utilizar este parámetro.

11.3 Trabajo futuro

El trabajo futuro más importante es la mejora del emulador del SID, especialmente la calidad de audio. Seguramente la mejor solución sea incluir un conversor D/A externo, permitiendo además aumentar la profundidad de muestra. También es importante implementar las funciones no implementadas del SID.

Otra mejora interesante sería generar las formas de onda directamente en función del valor del acumulador de fase, por modelado de ondas, tal y como hace el SID, en lugar de mediante una tabla de onda. Esto permitiría liberar mucha memoria en el emulador, dejando espacio libre para el almacenamiento de los coeficientes del filtro, facilitando su implementación. Actualmente los arrays de forma de onda ocupan la mayoría de la memoria RAM del Atmega168, 768 bytes siendo el tamaño total de 1024 bytes. El único límite para esta mejora es el tiempo de procesado necesario.

En cuanto al sintetizador en general, lo más importante es la mejora de la afinación fina, mediante una aproximación exponencial. Además, puede ser interesante aumentar sus funcionalidades: implementación de *pitch bend*, sensibilidad de tacto, implementación de LFOs internos que permitan modular los parámetros, arpegiador, etc. En este aspecto, hay muchas opciones posibles que aumentarían las posibilidades del sintetizador.

12. REFERENCIAS

Allan V. Oppenheim, Alan S. Willski, S. Hamid Nawab. 1998. *Señales y sistemas.* Segunda edición. Prentice Hall, 1998.

Arduino Playground. *MIDI Library v3.2*. [En línea] [Fecha de consulta: 2 de Junio de 2013.] http://playground.arduino.cc/Main/MIDILibrary.

Arduino playground. *SID-Emulator*. [En línea] [Fecha de consulta: 2 de Junio de 2013.] http://playground.arduino.cc/Main/SID-emulator.

Arduino. *Arduino - Homepage*. [En línea] [Fecha de consulta: 3 de Mayo de 2013.] http://www.arduino.cc/es/.

Atmel Corporation. 2009. *8-bit AVR Microcontroller with 4/8/16/32K bytes in-system programmable flash*. 2009. Disponible en internet [Fecha de consulta del enlace: septiembre de 2013]: http://www.atmel.com/images/atmel-8271-8-bit-avr-microcontroller-atmega48a-48pa-88a-88pa-168a-168pa-328-328p_datasheet.pdf

Banson. *SIDASTER* [Banson - Wiki]. [En línea] [Fecha de consulta: 26 de Mayo de 2013.] http://www.banson.fr/wiki/doku.php?id=sidaster.

Commodore semiconductor group. 1986. 6581 *SOUND INTERFACE DEVICE (SID)*. 1986. Disponible en internet [Fecha de consulta del enlace: septiembre de 2013]: http://www.datasheetarchive.com/dl/Scans-028/ScansU2X2400.pdf

Dekker, Ronald. *Flyback Converters for Dummies*. [En línea] [Fecha de consulta: 15 de Junio de 2013.] http://www.dos4ever.com/flyback/flyback.html.

Gratton, Angus. 2011. *The Hairless MIDI to Serial Bridge*. [En línea] 2011. [Fecha de consulta: 14 de Junio de 2013.] http://projectgus.github.io/hairless-midiserial/.

Haberer, Cristoph. *Atmega8 MOS6581 SID Emulator*. [En línea] [Fecha de consulta: 26 de Mayo de 2013.] http://www.roboterclub-freiburg.de/atmega_sound/atmegaSID.html.

HardSID. 1999. *HardSID. The 6581/8580 synthesizer*. [En línea] 1999. [Fecha de consulta: 6 de Junio de 2013.] http://www.hardsid.com/.

HVSC. *High Voltage SID Collection. Commodore 64 music for the masses.* [En línea] [Fecha de consulta: 18 de Junio de 2013.] http://www.hvsc.c64.org/#index.

Insert Piz here. *Insert Piz Here-vst plugins*. [En línea] [Fecha de consulta: 06 de Junio de 2013.] http://thepiz.org/plugins/.

Linear Technology. *LT1073. Micropower DC/DC converter, adjustable and fixed 5V 12V*. Disponible en internet [Fecha de conslulta del enlace: septiembre de 2013] http://cds.linear.com/docs/en/datasheet/1073fa.pdf

MacGateway. 2012. *The history of sound cards and computer game music*. [En línea] 13 de Julio de 2012. [Fecha de consulta: 7 de Mayo de 2013.] http://macgateway.com/featured-articles/sound-card-history/.

MIDI Manufacturers Association. 1995. *MIDI 1.0 Detailed Specification*. La Habra, California, 1995. Document Version 4.2.

O'Connell, Jaimie. *MIDI OX*. [En línea] [Fecha de consulta: 20 de Junio de 2013.] http://www.midiox.com/.

Sawsquarenoise. 2011. El *sonido en el Apple II*. [En línea] 8 de Diciembre de 2011. [Fecha de consulta: 6 de Junio de 2013.] http://www.sawsquarenoise.com/2011/12/el-sonido-en-el-apple-ii.html.

Trask, Simon. 1999. *Electron Sidstation sound module*. [En línea] Noviembre de 1999. [Fecha de consulta: 6 de Septiembre de 2013.] http://www.soundonsound.com/sos/nov99/articles/sidstation.htm.

Varga, Andreas. 1996. *Interview with Bob Yannes*. [En línea] Agosto de 1996. [Fecha de consulta: Junio de 2 de 2013.] http://sid.kubarth.com/articles/interview_bob_yannes.html.